gândirea algoritmică

elemente de bază ale limbajului Pascal

instrucțiunile de control

tipurile de date structurate

subprogramele

grafurile

structurile dinamice de date

fişierele

bibliotecile

grafica şi animația

programarea orientată pe obiecte

www.edusoft.ro

ISBN 973-87655-0-1

9 789738 765504

B. Pătruț - Învățați limbajul Pascal în 12 lecții

EduSoft

Bogdan Pătruț

Învățați limbajul Pascal în 12 lecții

EduSoft

Bogdan Pătruț

Învățați limbajul Pascal în 12 lecții

Editura EduSoft
Bacău – 2005

Redactor: Iulian Marius Furdu

Descrierea CIP a Bibliotecii Naţionale a României

PĂTRUŢ, BOGDAN
Învăţaţi limbajul Pascal în 12 lecţii / Bogdan Pătruţ. - Bacău : EduSoft, 2005
ISBN 973-87496-10-1

004.42 PASCAL

Copyright © 2005 Editura EduSoft
Toate drepturile asupra prezentei ediţii sunt rezervate Editurii EduSoft. Reproducerea parţială sau integrală a conţinutului, prin orice mijloc, fără acordul scris al Editurii EduSoft este interzisă şi se va pedepsi conform legislaţiei în vigoare.

Editura EduSoft
600065 Bacău, str. 9 Mai, nr. 82, sc. C, ap. 13
E-mail: contact@edusoft.ro, Web: www.edusoft.ro

ISBN **973-87496-10-1**

CUPRINS

Lecția 1. Introducere în informatică
1. Gândirea algoritmică
2. Elementele programării structurate
3. Variabilă. Constantă. Expresie. Instrucțiunea de atribuire
4. Limbaj de programare. Calculator electronic. Informatică
5. Tipuri de date
6. Exerciții recapitulative

Lecția 2. Elementele de bază ale limbajului
1. Identificatori. Constante
2. Variabilă. Tip de dată
3. Tipul `Integer`
4. Tipul `Real`
5. Tipul `Char`
6. Tipul `Boolean`
7. Tipul `String`
8. Operații comune tuturor tipurilor predefinite
9. Structura unui program în limbajul Pascal
10. Introducerea şi extragerea datelor
11. Utilizarea bibliotecii `Crt` pentru lucrul cu ecranul
12. Exerciții recapitulative

Lecția 3. Instrucțiuni de control
1. Instrucțiunea de decizie `IF`
2. Instrucțiunea repetitivă `WHILE`
3. Instrucțiunea repetitivă cu contor `FOR`
4. Instrucțiunea repetitivă cu test final `REPEAT`
5. Instrucțiunea de selecție multiplă `CASE`
6. Exerciții recapitulative

Lecția 4. Tipuri de date definite de programator
1. Tipul enumerare şi tipul subdomeniu
2. Tablouri unidimensionale (vectori)
3. Ordonarea şi interclasarea vectorilor
4. Alte aplicații ale vectorilor
5. Tipul `String`
6. Tablouri bidimensionale (matrice)
7. Tipul înregistrare
8. Tipul mulțime
9. Exerciții recapitulative

Lecția 5. Subprograme
1. Funcții şi proceduri
2. Aplicații ale subprogramelor
3. Variabile inițializate. Variabile de tip funcție sau procedură
4. Recursivitate
5. Aplicație: evaluator de expresii algebrice
6. Exerciții recapitulative

Lecția 6. Metode de elaborare a algoritmilor
1. Generarea de mulțimi
2. Metode simple de sortare
3. Metoda "*greedy*"

4. Metoda "*backtracking*"
5. Utilizarea mouse-ului în modul text
6. Metoda "*divide et impera*"
7. Metoda programării dinamice
8. Exerciții recapitulative

Lecția 7. Algoritmi referitori la grafuri

1. Noțiuni introductive
2. Explorarea grafurilor
3. Drumuri de cost minim în grafuri
4. Arborele parțial de cost minim
5. Exerciții recapitulative

Lecția 8. Structuri dinamice de date

1. Tipul referință
2. Stive şi cozi
3. Liste dublu înlănțuite
4. Arbori binari
5. Aplicație: derivare formală
6. Memorarea arborilor oarecare în arbori binari
7. Exerciții recapitulative

Lecția 9. Fişiere

1. Utilizarea fişierelor `DOS`
2. Generalități despre variabilele fişier
3. Fişlere text
4. Fişiere cu tip
5. Fişiere fără tip
6. Exerciții recapitulative

Lecția 10. Biblioteci

1. Generalități. Bibliotecile standard
2. Biblioteci create de programator
3. Aplicații

Lecția 11. Grafică şi animație

1. Noțiuni introductive
2. Figuri recursive
3. Graficul unei funcții
4. Pictură cu mouse-ul
5. Exploatarea fişierelor de caractere `CHR`
6. Animație. Fişierele `FLI`

Lecția 12. Programarea orientată pe obiecte

1. Introducere
2. Definirea tipurilor obiect
3. Derivare
4. Polimorfism
5. Metode virtuale
6. Un joc orientat obiect. Utilizarea *Sound Blaster* -ului
7. Exerciții recapitulative

Soluții

Bibliografie

Index

Introducere

La întrebarea firească a cititorului “De ce o nouă carte de Pascal” vom răspunde simplu: o nouă abordare, un nou stil de prezentare, un alt gen de aplicații.

Această carte se doreşte a fi o alternativă la manualele de Pascal, acoperind programa şcolară din clasele a IX-a şi a X-a. Prin modul de prezentare ales (dialog cu cititorul), considerăm că lucrarea poate fi parcursă cu succes (lecțiile 1-5 şi 11) şi de elevii claselor mai mici (a VII-a, a VIII-a). De asemenea, lecțiile 7, 8 şi 12 se adresează şi elevilor din clasele mai mari (a XI-a, a XII-a).

Aplicațiile prezentate în ultimele lecții ale cărții pot constitui un material de studiu interesant şi pentru studenți, profesori de informatică şi programatori experimentați.

Autorul a urmărit ca, pe parcursul celor 12 lecții, cartea să-l învețe pe cititor să lucreze eficient şi elegant într-unul din cele mai frumoase şi utilizate limbaje de programare.

Astfel, stilul de învățare este prin exemple şi exerciții.

Deşi toate lecțiile se dezvoltă în jurul unor elemente despre limbaj, pe care le-am prezentat gradat, totuşi există o structurare a cărții astfel:

- lecția 1 - un capitol introductiv care trebuie citit cu foarte mare atenție, deoarece noțiunile prezentate intuitiv aici se regăsesc pe cuprinsul întregii lucrări.
- lecțiile 2-5 - prezentarea elementelor de bază ale limbajului (tipuri de date, instrucțiuni, subprograme) - exemplele alese sunt simple, punându-se accent pe elementele de limbaj şi mai puțin pe cele de algoritmicâ.
- lecțiile 6-8 - prezentarea unor tehnici de programare eficientă, în paralel cu elementele de limbaj care constituie suportul aplicațiilor; se pune accent pe algoritmi şi pe modelarea datelor.
- lecțiile 9-12 - aplicațiile prezentate în aceste lecții sunt foarte complexe, necesitând un plus de experiență din partea programatorului.

În carte au fost incluse o serie de aplicații care se referă la diferite echipamente periferice (mouse, ecran, tastatură, imprimantă, Sound Blaster). Acestea se adresează programatorilor avansați, dar şi tuturor celor ce doresc să învețe repede cum se poate exploata un calculator din toate punctele de vedere.

Doresc să mulțumesc pe această cale familiei mele, precum şi tuturor celor ce m-au încurajat şi ajutat în munca mea de a scrie şi pune la punct această lucrare: Liliana Dăbuleanu, Adrian Atanasiu, Tudor Sorin, Clara şi Eugen Ionescu, Liliana Crăciun.

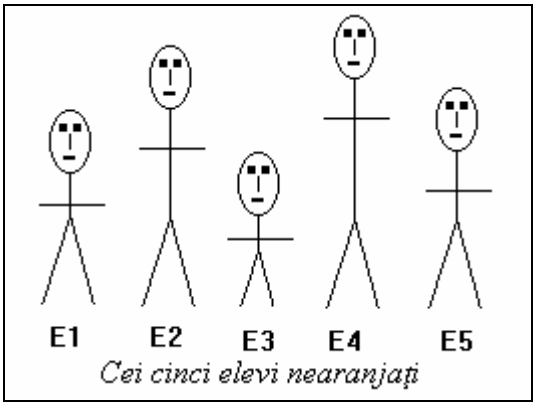
Iar cititorilor nu-mi rămâne decât să le urez lectură plăcută.

Autorul,
Bacău, 24 iunie 1996. – oct. 2005

Lecția 1
INTRODUCERE ÎN INFORMATICĂ

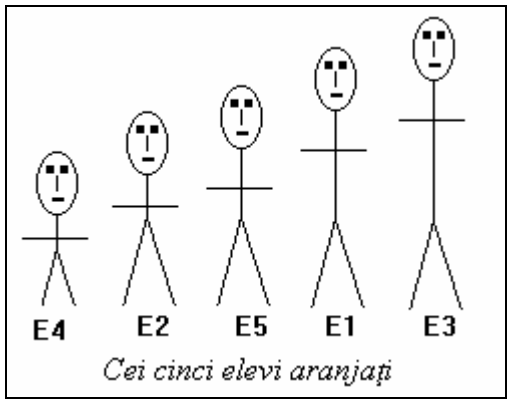
1. Gândirea algoritmică

Problema ordonării crescătoare.
Avem un şir de 5 elevi, de diferite înălțimi, şi ne propunem să-i reaşezăm în ordinea crescătoare a înălțimilor:



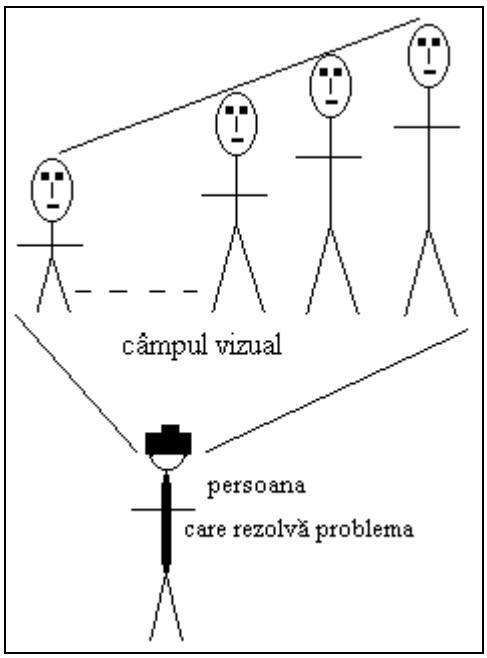
Cei cinci elevi nearanjați

O persoană care nu cunoaşte *gândirea algoritmică* îi va aşeza cu uşurință în ordinea crescătoare dorită, privindu-i pe toți şi făcând câteva comparații ale înălțimilor lor. În final, elevii vor fi aşezați ca în figura următoare:



Cei cinci elevi aranjați

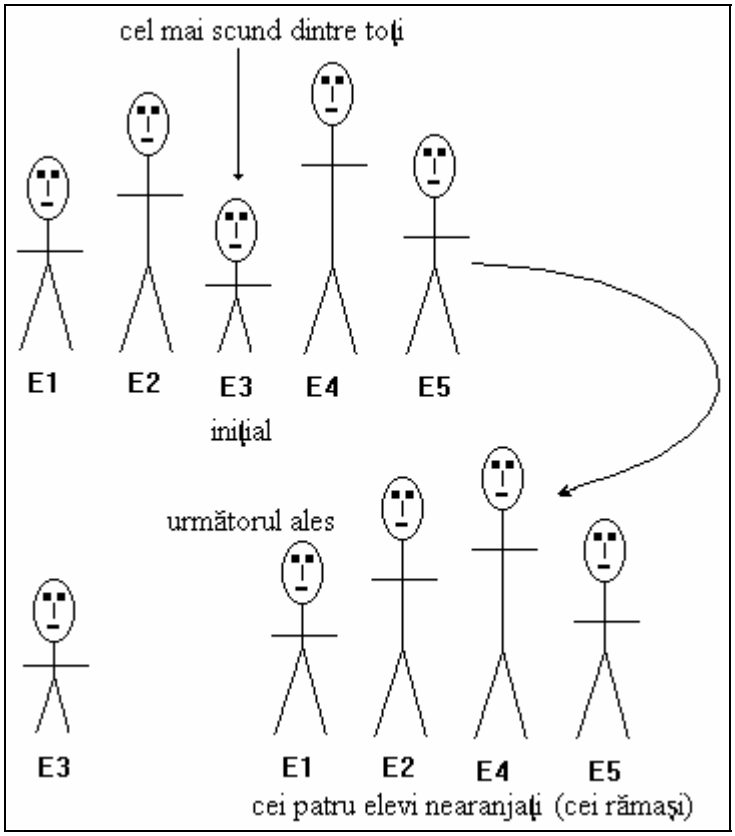
Să considerăm, însă, că numărul elevilor este un milion. De această dată, persoana care va rezolva problema va fi pusă în situația dificilă de a privi deodată toți cei un milion de elevi, pentru a-i compara între ei. În plus, în nici un moment nu va putea fi sigură că elevii sunt puşi în ordinea crescătoare a înălțimilor lor, decât dacă le poate privi vârfurile capetelor tuturor:



Chiar dacă va reuşi, îi va veni foarte greu să facă această ordonare care, probabil va dura foarte mult. De aceea, într-o astfel de problemă trebuie să se aplice *o tehnică generală, un procedeu bine stabilit*, care să permită ordonarea crescătoare, în timp util, a oricâtor elevi, indiferent de numărul lor sau de aşezarea lor inițială. Se constată că, indiferent care va fi procedeul sau *strategia* care se va adopta, acesta va necesita efectuarea unor *comparații între câte doi elevi*.

Iată o metodă generală de soluționare a problemei date: se ia cel mai scund dintre elevi şi se aşază deoparte, apoi se procedează la fel cu restul elevilor; adică se ia cel mai scund din cei rămaşi şi se aşază alături de primul scos din rând şi tot aşa. Metoda soluționează problema, dar ea implică rezolvarea unei subprobleme (probleme mai mici): alegerea unui cel mai scund elev din mai mulți, puşi la rând.

Problema determinării minimului.
Ultima problemă pusă mai înainte se rezolvă după cum urmează: se consideră primul elev din rând ca fiind, până la proba contrarie, cel mai scund elev. Se parcurge rândul de elevi şi, dacă se întâlneşte un elev mai scund, se renunță la primul, declarând noul elev drept cel mai scund; cu acesta se procedează la fel, în continuare. Fireşte, pentru a decide care din doi elevi este mai mic în înălțime, se vor compara înălțimile lor. Iată cum am depistat, mai sus, că elevul `E3` e cel mai scund din lista `E1`, `E2`, `E3`, `E4`, `E5`: Îl considerăm pe `E1` (deci primul din şir, de la stânga la dreapta) drept cel mai mic. Îl comparăm pe `E1` cu `E2`. Observăm că `E1` e mai mic decât `E2` (să notăm: `E1<E2`). Atunci îl păstrăm pe `E1` (deci îl considerăm tot cel mai mic dintre toți), renunțăm la `E2` şi trecem la următorul, deci la `E3`. Comparându-l pe `E1` cu `E3`, observăm că `E3<E1`, deci am dat peste proba contrarie pentru `E1`, deci va trebui să-l considerăm pe `E3` cel mai mic, de-acum încolo.



Deci, în continuare, îl vom compara pe `E3` cu cei de după el (deci cu `E4` şi `E5`) şi vom observa că `E3` rămâne cel mai scund, deci el este cel mai scund. Pe `E3` nu-l comparăm cu elevii dinaintea sa, deoarece înaintea lui `E3` aveam elevul `E1`, care era mai cel mai mic printre ei, iar `E3<E1`. (În exemplul nostru, cum `E1<E2`, iar `E3<E1`, rezultă că avem şi `E3<E2`). Aşadar, ajunşi la `E3`, privim tot înainte, fără a ne mai întoarce înapoi.

Să recapitulăm, deci, tehnica de ordonare (sortare) crescătoare a elevilor, după înălțimile lor. Vom căuta să o descriem cât mai simplu şi sugestiv:

```
ordonarea a 5 elevi înseamnă:
început
    alegerea celui mai scund elev din cei 5 elevi;
    aezarea celui mai scund elev într-un nou sir;
    alegerea celui mai scund elev din cei 4 elevi rămasi;
    asezarea celui mai scund elev în noul sir;
    ...........
    asezarea celui mai scund elev în noul sir
sfârsit.
```

De fapt, metoda descrisă se compune din:

```
început
crearea unui nou sir de elevi, care la început este fără elevi,
      urmată de:
        alegerea celui mai scund elev din cei 5 elevi;
```

```
        adăugarea celui mai scund elev în noul sir;
        ordonarea celor 4 rămasi
sfârsit.
```

Fireşte, ordonarea celor 4 elevi rămaşi se va face la fel ca şi a celor 5:

```
ordonarea celor 4 elevi înseamnă:
început
        alegerea celui mai scund elev din cei 4 elevi;
        adăugarea celui mai scund elev în noul sir;
        ordonarea celor 3 elevi rămasi
sfârsit.
```

Aşadar, în general, ordonarea crescătoare se realizează astfel:

```
început
creează un nou sir, initial vid;
ordonează cei n elevi, care înseamnă
   început
        alegerea celui mai scund elev din cei n;
        adăugarea celui mai scund elev în noul sir;
        ordonează cei n-1 elevi rămasi
   sfârsit
sfârsit.
```

Pentru descrierea acțiunilor ce trebuiesc realizate am folosit propoziții în limba română; acestea fac anevoioasă descrierea procedeului, motiv pentru care ne rezumăm la cuvintele cele mai semnificative:

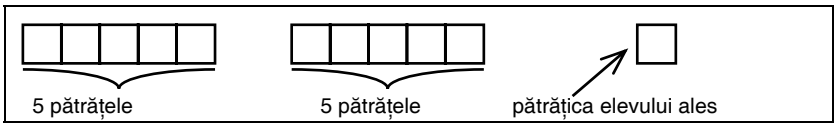
```
aranjare(n)înseamnă
început
   creare_sir_vid;
   ordonare(n)
sfârsit.

ordonare(n)înseamnă
început
   alegere_scund(n); adaugă_scund(n); ordonare(n-1)
sfârsit.
```

Vom nota acest procedeu prin (`A`).
Aparent, totul este în regulă, dar când `n` este (sau devine) `1`, apare întrebarea: “ce înseamnă `ordonare(1-1)`, adică `ordonare(0)`? ”. Operația de ordonare nu are sens decât dacă `n>0`:

```
ordonare(n) înseamnă:
început
   dacă n>0 atunci
        început
          alegere_scund(n); adăugare_scund; ordonare(n-1)
        sfârsit
sfârsit.
```

Să revenim la procedeul (`A`). Primul pas era crearea şirului vid de elevi. Ne putem imagina această operație prin rezervarea unui spațiu, în altă parte decât pe locul unde sunt situați elevii nearanjați încă. Acel spațiu se va umple, pe rând, cu câte un elev, până când nu vor mai rămâne deloc elevi în spațiul inițial. Aşadar, avem nevoie de 10 mici suprafețe pătrate, pe care să stea cei 5 elevi:



pentru pozițiile finale	pentru pozițiile inițiale	(cel mai scund curent)

Acestor 10 pătrățele li se mai adaugă una specială, acolo unde va fi aşezat elevul cel mai scund (dintre cei rămaşi), pentru a trece dintr-o parte în alta. Deci pentru `n` elevi vom avea nevoie de `2n+1` pătrățele. Dacă de fiecare dată elevul selectat va fi pus după cei ordonați deja şi înaintea celor neordonați, folosind aceleaşi `n` pătrățele, atunci soluția este mai economă, deci vor fi necesare doar `n+1` pătrățele. Aşadar, aceeaşi problemă se poate rezolva folosind *mai puțin spațiu*.

Problema căutării.
Există şi cazuri în care, pentru o aceeaşi problemă putem prezenta două soluții, în care una este *mai rapidă* ca alta. De pildă, fie un şir de numere naturale oarecare, de exemplu `4,2,10,1,8,15,7`. Vrem să testăm dacă un număr dat (să zicem numărul `8`) se află în această secvență sau nu.

Dintr-o privire observăm că răspunsul este afirmativ, dar nu ne-am descurca prea uşor cu un milion de numere, căci n-am putea privi un milion de numere *deodată*. Deci, să le privim pe rând. Dar de unde să începem? Cum nu există un capăt preferențial, să începem din stânga. Primul număr întâlnit este `4`. Comparăm acest număr cu numărul căutat (`8`) şi vedem că diferă. Trecem mai departe. `2` nu e nici el egal cu `8`, apoi nici `10`, nici `1`, dar în sfârşit `8` (cel din listă) este egal cu `8`(cel căutat). Nu are sens să continuăm, deci ne oprim aici. Fireşte, cel mai defavorabil caz ar fi fost ca `8`-ul din şir să fi fost pe ultima poziție. Deci, pentru `n` numere, în cel mai defavorabil caz am face `n` "priviri", deci `n` comparații.

O astfel de căutare, prin parcurgerea de la stânga la dreapta a întregii secvențe de numere, până se găseşte numărul dorit sau se epuizează toate elementele din secvență se numeşte **căutare secvențială**. Dacă avem un şir `S` de `n` elemente (notat `S[1..n]`), atunci căutarea secevnțială a lui `x` în `S` se descrie prin:

```
căutare_secventială(x,S[1..n]) înseamnă
început
   fie elementul_curent = primul_element;
   atât timp cât (elementul_curent ≠ x) si
        (pozitia elementului curent ≤ pozitia ultimului element (n))
        execută început
                  dacă elementul_curent = x atunci mesaj('găsit')
                  altfel treci la următorul element
                sfârsit
sfârsit.
```

În continuare să presupunem că numerele erau *deja ordonate crescător*: `1,2,4,7,8,10,15`. În acest caz particular, căutarea secvențială a unui număr cum e `8` nu e prea eficientă, deoarece `8` se află în a doua jumătate a secvenței, deci ar fi de preferat să nu-l căutăm în prima jumătate. Pe poziția din centru este `7`, iar în prima jumătate vor fi doar numere mai mici decât `7`, deci mai mici şi decât `8`. De aceea, îl vom căuta pe 8 în acea jumătate în care, în mod logic, *s-ar putea găsi*. Acest procedeu este mai rapid.

♦ dacă numărul din mijloc este mai mic decât numărul căutat, atunci căutăm în a doua jumătate;

♦ dacă numărul din mijloc este mai mare ca numărul căutat, atunci căutăm în prima jumătate;

♦ dacă numărul din mijloc este egal cu numărul căutat, înseamnă că am găsit numărul în cauză si trebuie să oprim căutarea.

Căutarea în jumătatea aleasă se face tot la fel, deci se va înjumătăti si această zonă etc..

Procedeul anterior se numeşte **căutare binară**, deoarece, de fiecare dată, o secvență de numere este divizată în două jumătăți. Putem descrie căutarea binară a unui număr `x` într-o secvență `S[1..n]` astfel:

```
căutare_binară(x, S[1..n]) înseamnă
început
  stabileste elementul_curent ca fiind elementul din mijlocul secventei;
  dacă n=1 atunci
       dacă x = elementul_curent atunci mesaj('găsit')
       altfel mesaj('negăsit')
  altfel
   început
    împarte secventa în cele două jumătăti (S[1..mijloc] si S[mijloc+1..n]);
    dacă elementul_curent = x atunci mesaj('găsit')
    altfel
      dacă elementul_curent < x atunci căutare_binară(x,S[mijloc+1..n])
      altfel căutare_binară(x,S[1..mijloc])
   sfârsit
sfârsit.
```

Metodele sau procedeele de rezolvare a unei probleme pe un caz general, grupate sub un acelaşi nume şi având o structură de tipul celor descrise până acum, le vom numi **proceduri**.

Observăm că procedura de căutare binară se foloseşte de ea însăşi, pentru dimensiuni mai mici ale problemei, deci *se autoapelează*. O astfel de procedură se numeşte **recursivă**.

O trăsătură fundamentală a procedurilor (recursive sau nu) este că, pe lângă cuvintele cu rol general (*început, sfârşit, înseamnă*) cuprind şi nişte grupuri de cuvinte "speciale":

- `dacă...atunci...altfel...` (sau chiar numai: `dacă...atunci...`)
- `atât timp cât...execută...` (sau `cât timp...execută (repetă)...`)

Observăm că procedeele descrise reprezintă *succesiuni* de acțiuni (operații). Acestea se execută una după alta, de sus în jos şi se numesc **instrucțiuni**. Respectându-le, orice persoană va putea rezolva cu uşurință problemele în cauză.

În plus, rezolvitorul va avea două certitudini asupra unei astfel de rezolvări:

- *se termină* (în timp finit, oricât de mare ar fi acesta);
- *este corectă* (adică rezolvă problema pe orice caz, indiferent de dimensiunea sa).

Întotdeauna trebuie să urmărim ca procedeul de rezolvare a unei probleme:

1. să se termine;
2. să fie corect;
3. să dureze cât mai puțin;
4. să necesite cât mai puțin spațiu.

Un procedeu constituit dintr-o succesiune de instrucțiuni, descris clar, fără ambigutăți, care îndeplineşte (pentru o clasă de probleme dată) primele două condiții de mai sus, se numeşte **algoritm** (de rezolvare a respectivei clase de probleme). Modul de gândire folosit în scrierea de algoritmi se numeşte **gândire algoritmică**. Un algoritm care îndeplineşte şi condiția 3 poate fi considerat un *algoritm bun*, iar dacă îndeplineşte şi condiția 4, fără să o afecteze pe 3, atunci este un *algoritm foarte bun*. Persoana care elaborează algoritmi de rezolvare pentru diferite (clase de) probleme se numeşte **programator**. Un algoritm poate fi reprezentat fie de o singură procedură, fie din mai multe, grupate şi apelate pe rând. Un astfel de grup se numeşte **program**.

Astfel de programe omul execută câteodată chiar fără să-şi dea seama. De exemplu, atunci când vrem să căutăm o pagină într-o carte, nu luăm pagină cu pagină, până o găsim (căutare secvențială), ci facem aproximativ o căutare binară: deschidem cartea undeva la

mijloc, vedem ce număr are pagina de acolo şi, în funcție de el, ne îndreptăm atenția spre paginile din față sau spre cele din spate, pentru care procedăm la fel.

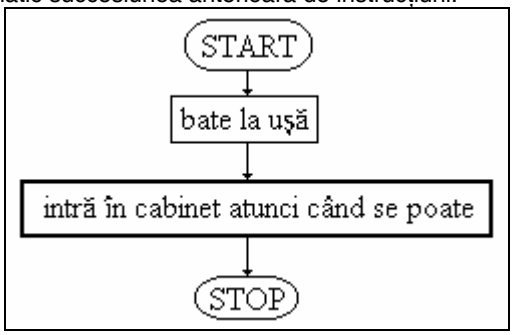
2. Elementele programării structurate

Spuneam că o caracteristică importantă a procedurilor este utilizarea grupurilor de cuvinte: **dacă...atunci...altfel...** şi **atât timp cât...execută...** .
De asemenea, precizam că într-o procedură, instrucțiunile se execută una după alta, de sus în jos, adică *secvențial*. Vom da un exemplu de procedură care foloseşte toate aceste elemente.

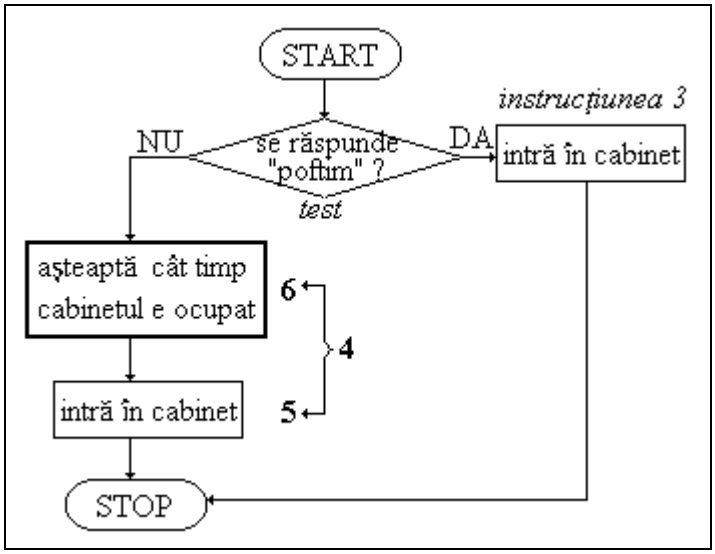
Problema intrării într-un cabinet medical.
Dacă un om vrea să intre pentru consult la un medic, atunci, mai întâi va bate la uşă; dacă medicul răspunde prin "poftim !", atunci va intra, altfel va aştepta până când pacientul dinăuntru va ieşi; Abia după ce cabinetul va fi liber, va intra.

```
   intrare_la_medic înseamnă
   început
1.   bate_la_usă;
2.  dacă răspunsul = 'poftim !' atunci
    3.   intră_în_cabinet;
    altfel
         început
    4.        atât timp cât cabinetul_este_ocupat_de_alt_pacient
                  execută
    6.              citeste_un_ziar;
    5.        intră_în_cabinet
         sfârsit
   sfârsit.
```

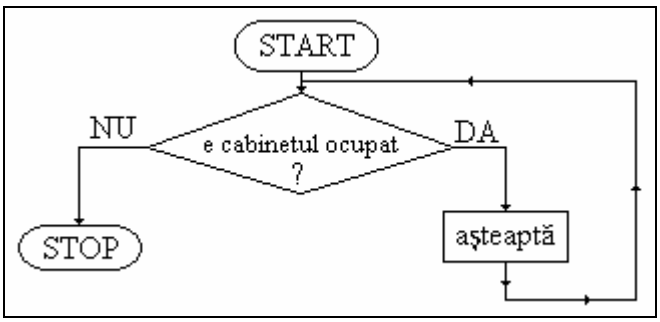
Putem reprezenta schematic succesiunea anterioară de instrucțiuni:



Primul dreptunghi corespunde instrucțiunii 1, al doilea instrucțiunii mari 2. Aceasta este compusă din mai multe instrucțiuni: 3,4,5; instrucțiunea 4 este compusă din anumite cuvinte speciale şi din instrucțiunea 6. De aceea am desenat îngroşat cel de al doilea dreptunghi. Instrucțiunea respectivă se detaliază astfel:

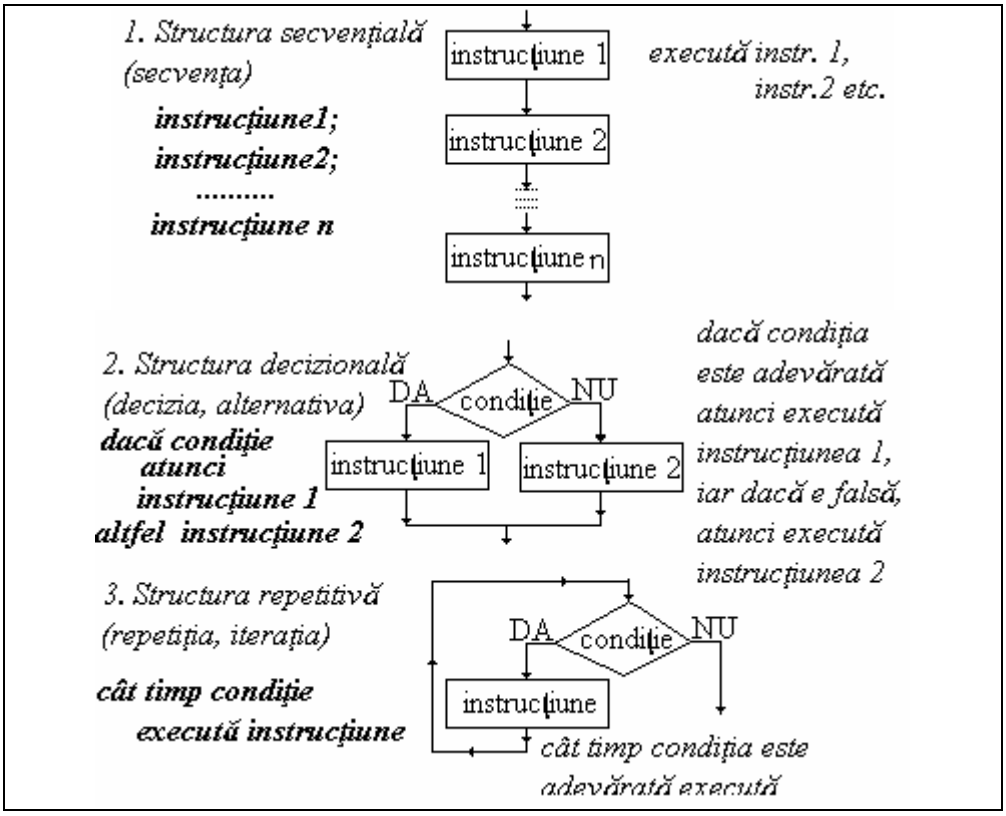


Aşadar, condiția încadrată de cuvintele `dacă` şi `atunci` este reprezentată în schema anterioară printr-un romb, cu o intrare şi două ieşiri spre cele două ramuri: `NU` şi `DA`. Ramura `NU` conține, la rândul ei o instrucțiune complexă:



Çi instrucțiunea `atât timp cât...execută...` conține o condiție dintr-un romb. Ea este încadrată de grupurile de cuvinte `atât timp cât` şi `execută`.

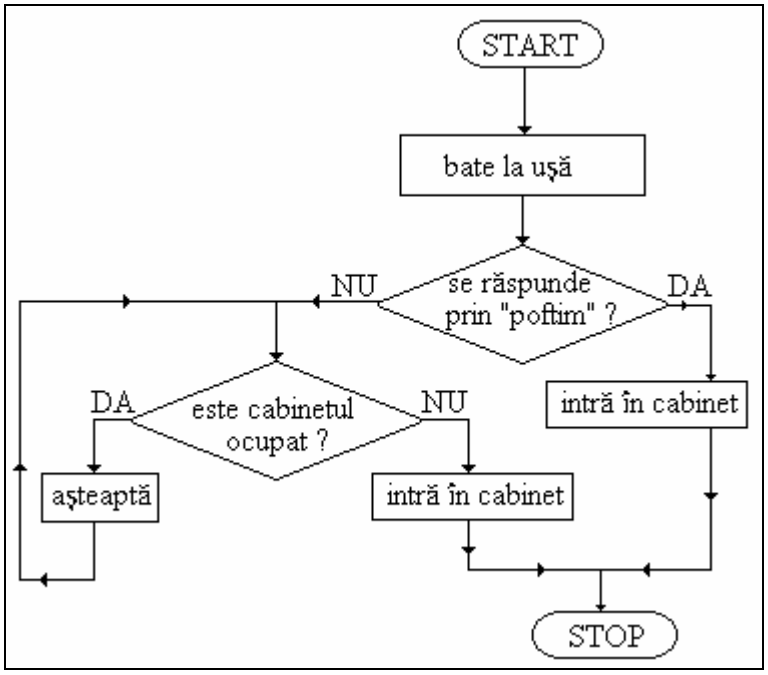
Aşadar, există trei **structuri** importante ce ne permit să descriem schematic procedurile (algoritmii):



Fireşte, prin `instructiune` se poate înțelege, în toate cele trei cazuri, fie o instrucțiune simplă, fie una compusă.

O **instrucțiune compusă** este formată dintr-o *secvență* de instrucțiuni, încadrate de cuvintele `inceput` şi `sfârsit`, şi poate conține, la rândul lor, blocuri de *alternativă şi repetiție*.

Schema completă a problemei intrării în cabinetul medical va fi:



O astfel de schemă grafică, realizată din romburi, cu o intrare şi două ieşiri, din dreptunghiuri, cu o intrare şi o ieşire, precum şi din dreptunghiurile racordate `START` (obligatoriu unic), cu o ieşire, şi `STOP`, fără ieşire şi cu cel puțin o intrare, se numeşte **schemă logică**. Folosirea lor pentru a descrie un algoritm *nu este recomandată*, deoarece, utilizând structurile secvențială, alternativă şi repetitivă, se pot descrie algoritmi mult mai inteligibil prin cuvinte. **Programarea** este tehnica realizării de algoritmi descrişi prin proceduri şi programe. Ea devine o artă, atunci când se folosesc cele trei elemente de structurare şi se numeşte **programare structurată**.

Pentru a vedea ce ar însemna programare nestructurată, să reconsiderăm procedura de intrare în cabinetul medical:

```
intrare_la_medic înseamnă
început
   bate_la_usă;
   dacă răspunsul = 'poftim !' atunci intră_în_cabinet;
   altfel început
            atât timp cât cabinetul_este_ocupat_de_alt_pacient execută
                 citeste un ziar;
             intră_în_cabinet
          sfârsit
sfârsit.
```

Instrucțiunea încadrată mai sus se mai poate scrie şi astfel:

```
A       dacă cabinetul este ocupat atunci
        început
            citeşte_un_ziar;
            treci la pasul A
        sfîsit
```

Această întoarcere ("treci la pasul `A`") (care seamănă cu întoarcerile din schemele logice) deranjează deoarece trecerea la pasul anterior `A` înseamnă *nerespectarea* regulii conform

căreia într-o procedură *instrucțiunile se execută una după alta, de sus în jos*. Iată cel puțin un motiv pentru care o astfel de instrucțiune, numită **instrucțiunea de salt necondiționat,** *nu trebuie folosită*.

Dacă saltul ar fi către o instrucțiune foarte îndepărtată, atunci ar fi mai greu de urmărit procedura. Informaticienii au demonstrat că nici *nu e nevoie* să fie folosită această instrucțiune de salt, deoarece cele trei elemente ale programării structurate sunt *suficiente* pentru a descrie orice procedură. Dacă, însă, folosirea instrucțiunii de salt ar face procedura mai lizibilă, atunci se poate apela la această instrucțiune.

3. Variabilă. Constantă. Expresie. Instrucțiunea de atribuire

Pentru a ordona crescător nişte elevi după înălțimile lor, trebuie să-l alegem, la un moment dat, pe *cel mai scund* din şirul de elevi rămaşi neordonați. Iar când vorbim de *cel mai scund*, nu ne interesează decât înălțimile elevilor, nu şi cât sunt de slabi sau de graşi, de frumoşi sau de urâți sau cum îi cheamă. Aşadar, fie `a,b,c` înălțimile a trei elevi şi vrem să determinăm care elev este cel mai scund, ceea ce înseamnă să determinăm *minimul* celor trei numere. Mai întâi, să notăm (în loc de `a`, `b` şi `c`), prin h_1, h_2 şi h_3 înălțimile celor trei elevi. Deci prin h_i vom înțelege, în general, înălțimea unui elev oarecare, al `i`-lea în şirul de elevi; aici `i` este un număr între `1` şi `3`. În matematică, foloseam litere (`a,b,c, x,y,z, f,g`) pentru a nota constante, variabile (necunoscute) şi funcții. Noi vom folosi şi *grupuri de litere şi cifre* pentru a nota aceste elemente din matematică. Astfel, în loc de `a,b,c` este mai clar să notăm cu `h1,h2,h3` înălțimile celor trei elevi, sau cu indici, h_1, h_2 şi h_3, după cum am arătat. De asemenea, formula de calcul a vitezei: `v=d/t` este mult mai sugestivă dacă se scrie astfel: `viteza=distanta/timp`. În cazul nostru, pentru a nota înălțimea celui mai mic elev este preferabil să folosim: `minim` (sau `mic` sau `scund)` şi nu `m` sau `s`. Notația `minim` pentru desemnarea înălțimii celui mai scund elev poate fi folosită şi pentru a desemna minimul dintr-o mulțime de numere oarecare. Aceste numere pot fi asociate oricăror gen de măsuri: centimetri (reprezentând înălțimile unor elevi), kilograme (reprezentând cantități) etc..

Pentru a determina minimul a `n` numere procedăm aşa cum spus şi în prima secțiune: considerăm primul număr minim şi, până la proba contrarie, el rămâne minimul. Dacă, însă, găsim alt număr mai mic, atunci acesta e considerat minimul. Facem aşa până la ultimul număr. Deci, având `n` numere h_1, h_2 , ..., h_n:

```
        determinare_minim(n) înseamnă
        început
1.              fie i = 1;
2.              fie minim = hi ;
                atât timp cât i < n execută
                        început
                            dacă hi < minim atunci
3.                              fie minim = hi
4.                          altfel fie i = i + 1
                        sfârsit
        sfârsit.
```

Să explicăm cele patru instrucțiuni numerotate, care au o caracteristică comună. Când spunem `fie i=1`, înseamnă că îl facem pe `i` egal cu `1`. Deci, în continuare, orice referire la `i`, va însemna o referire la valoarea `1`. De aceea, prin `fie minim=`h_i se înțelege că minim va deveni h_i, deci (cum `i=1`), `minim` va deveni h_1.

În grupul `3`, trebuie să ştim că h_i este cunoscut (el este ori h_1, ori h_2, ..., ori cel mult h_n). Dacă acest h_i curent este mai mic decât elementul pe care îl consideram minim, până acum, atunci va trebui să reconsiderăm minimul. Astfel, `minim` va deveni, în continuare, h_i .

Dacă nu se întâmplă cele de mai sus (deci `minim` rămâne acelaşi), atunci executăm `4.fie i=i+1`. Pare ciudat, dar `i=i+1` nu înseamnă o relație matematică, ci pur şi simplu faptul că `i` se măreşte cu `1`. Astfel, când spuneam `fie i=1` însemna că lui `i` îi dăm valoarea `1`, când spuneam `fie minim=h`$_i$ însemna că lui `minim` îi dăm valoarea lui `h`$_i$, deci, în general, dacă spunem "`fie...=...`" înseamnă că ce este în stânga primeşte valoarea a ceea ce este în dreapta (fără ca ceea ce este în dreapta să-şi modifice valoarea). Aşadar, are sens `fie i=i+1`, pentru că în dreapta avem o valoare cunoscută. Dacă `i=3`, de exemplu, atunci `i+1` va fi `4`, deci `fie i=i+1` echivalează cu a spune `fie i=3+1`, deci `fie i=4`.

În continuare, orice instrucțiune de forma `fie x=e` va fi notată prin `x:=e`. Ceea ce este în dreapta este, într-adevăr. un "`x`", adică ceva care se schimbă, primind o nouă valoare. `x` se numeşte, de aceea, **variabilă**. Pe de altă parte, `e` desemnează o **expresie (**atât `1`, cât şi `h`$_i$ sau `i+1` sunt expresii matematice (`i` este un **nume de variabilă**). `1` e o **constantă** (ca şi `3`, `100.45`, `-24`), deci e o **expresie constantă numerică**, `h`$_i$ este un **nume de variabilă indexată** (în programare se notează prin `h[i]`), deci este o expresie, iar `i+1` este tot o expresie aritmetică, aşa cum şi `x+1`, `2*n+1`, `x+y*5-(i+j)/k` sunt tot expresii (simbolurile `*` şi `/` notează înmulțirea, respectiv împărțirea).

Prin `x:=e` înțelegem că lui `x` i se atribuie valoarea expresiei `e`, iar o astfel de instrucțiune se numeşte **instrucțiune de atribuire** (sau **asignare**).

Pentru a vedea cum funcționează atribuirea pe un caz concret, să considerăm o ecuație de gradul I, scrisă sub forma `ax+b=c`. Ea se rezolvă astfel: `ax+b=c |-b` ⇒ `ax+b-b=c-b` ⇒ `ax=c-b |: a` ⇒ `x=(c-b)/a`. Fireşte, ultima operație, de împărțire, are sens doar dacă `a`≠`0`. Formula finală şi această condiție ne conduc la:

```
rezolvare_ecuatie(a, b, c) înseamnă
început
        dacă a ≠ 0 atunci x : = (c - b ) / a
        altfel mesaj ('Nu se poate împărti prin zero !')
sfârsit.
```

Iată că formula `x=(c-b)/a` devine (în algoritmul descris) instrucțiunea de atribuire `x:=(c-b)/a` (apare **:=** în loc de **=**).
În afara celor 3 elemente ale programării structurate, instrucțiunea de atribuire şi noțiunile de constantă, variabilă şi expresie sunt *elementele esențiale ale programării*.

4. Limbaj de programare. Calculator electronic. Informatică.

- Având o sintaxă foarte simplă, limba engleză s-a impus în lumea ştiinşifică, astfel încât şi în domeniul nostru ea este aproape singura utilizată. Traducând în engleză termenii învățați în paragraful despre programarea structurată, avem:
  `atât timp cât ...` `execută` → **`while ... do ...`**
  `dacă ... atunci ... altfel ...` → **`if ... then ... else ...`**
  `precum si început` → **`begin ;`** `sfârsit` → **`end .`**

Procedura autoapelantă de ordonare se poate scrie acum:

```
ordonare (n) înseamnă
begin
  if n > 0 then
     begin
        alegere_scund(n); adăugare_scund;
         ordonare(n-1)
```

```
        end
    end.
```

- Convenim să renunțăm la cuvântul "înseamnă". Faptul că urmează descrierea procedurii ordonare se va preciza folosind cuvântul "procedure", la început:

```
procedure ordonare(n) ;
begin
       if  n>0 then
           begin
              alegere_scund(n) ; adăugare_scund ;
              ordonare(n-1)
           end
end ;
```

- Observăm că am folosit, simbolul "**;**" în două cazuri diferite: pentru a separa cele două apeluri de alte proceduri (care sunt nişte instrucțiuni) precum şi pentru a marca sfârşitul descrierii procedurii de ordonare (în loc de punct) sau pentru a delimita antetul ei (în loc de "înseamnă").

- Dacă noi am scrie această procedură pe o foaie de hârtie şi i-am da-o cuiva să o execute, pentru un anumit n, el va ajunge la apelul alegere_scund(n), deci va trebui să apeleze o altă procedură, pe care ar trebui să o ştie deja. De aceea, e bine ca procedura alege_scund să fie descrisă undeva înainte, pe foaia de hârtie. La fel ar trebui să se întâmple şi cu adăugare_scund.

Iată, deci, câteva **convenții folosite în descrierea procedurilor**:
1. Dacă o procedură P foloseşte o procedură Q, atunci mai întâi se scrie Q, apoi P.
2. Orice descriere a unei proceduri începe cu cuvântul procedure, urmat de numele procedurii şi, în paranteze, argumentele sale; noțiunea de *argument* va fi explicată în lecția 5. Argumentul unei proceduri este similar celui a unei funcții: x pentru f(x).
3. instrucțiunile dintr-o procedură, inclusiv la început şi la sfârşit, sunt despărțite de simbolul *punct şi virgulă* "**;**" .
4. Procedurile se pot autoapela.
5. în descrierile procedurilor se folosesc cuvintele "begin" şi "end" pentru a marca corpul acesteia.

Observăm, aşadar, utilizarea unor *reguli de scriere* a procedurilor. Descrierile procedurilor utilizează, de asemenea, *o mulțime de cuvinte* englezeşti: begin, end, procedure, if, then, else, while, do. Deci procedurile se descriu conform unor reguli asemănătoare regulilor de *sintaxă* adică de scriere a propozițiilor într-o limbă. Ele utilizează un *vocabular* adecvat. *Sintaxa şi vocabularul* constituie un *limbaj*. Cum acest limbaj ne permite să descriem proceduri (programe), el se poate numi **limbaj de programare**.

Dacă vom urmări cu atenție, vom observa că limbajul nostru (unii autori numesc un astfel de limbaj - **pseudocod**) are unele deficiențe. Una ar fi aceea că, de exemplu, nu putem lucra decât cu numere; astfel, când scriam "procedure ordonare(n)**;**" intuiam că n este un număr natural, dar un limbaj mai evoluat ar preciza acest lucru.

De-a lungul anilor, au fost create şi perfecționate mai multe limbaje de programare. Mai mult chiar, s-au creat *maşini electronice automate* care să poată "citi" programe scrise în astfel de limbaje şi să le execute întocmai. Aceste maşini poartă denumirea de **calculatoare electronice**. Ele pot realiza nu doar calcule matematice, ci şi alte operații, de pildă de sortare şi căutare.

Practic, în aranjarea crescătoare a elevilor, ceea ce ne interesează şi trebuie "spus" calculatorului este că elevii există, sunt în număr de cinci şi că ei au înălțimile pe care le au.

Acestea se numesc **informații**. În urma aplicării algoritmului dat de noi, calculatorul reuşeşte să ne prezinte lista elevilor în ordinea crescătoare a înălțimilor lor. Prin urmare el s-a folosit de algoritmul nostru de sortare pentru a *face ceva cu informațiile primite inițial*. Adică *a prelucrat* informațiile. Noi nu am intervenit în nici un fel în timpul prelucrării. Deci calculatorul electronic *prelucrează automat informațiile date, în conformitate cu un algoritm dat de om*.

Francezii au reunit denumirile de "*information*" (*informație*) şi "*automatique*" (*automat*) într-un singur termen "*informatique*" care în românește a devenit **informatică**, termen ce denumeşte, aşadar, *prelucrarea automată, cu ajutorul calculatoarelor electronice programate, a informațiilor*, tocmai disciplina sau, mai bine zis, ansamblul de discipline în tainele căruia încercăm să pătrundem împreună, pe parcursul acestor lecții.

5. Tipuri de date.

Într-o declarație de forma `procedure ordonare(n)`, n reprezintă argumentul procedurii; adică procedura `ordonare` ordonează `n` elevi, unde `n` se va preciza ulterior. Noi convenim ca `n` să fie număr natural, chiar dacă din declarația de mai sus nu rezultă acest lucru.

În limbajele evoluate de programare, fiecare argument, fiecare variabilă are un anumit **tip** bine definit, adică poate lua valori dintr-o mulțime precizată de valori.

De pildă, acel `n` din `ordonare(n)` este un număr natural, dar putem considera că ar fi un număr întreg sau un număr real, caz în care nu ar mai avea sens procedura. Când o variabilă `x` poate lua valori din mulțimea `tip` vom scrie `x:tip`. În algoritmii simpli, putem folosi următoarele tipuri de date: `Integer` = mulțimea numerelor întregi, `Real` = mulțimea numerelor reale, `Char` = caracter, `String` = şir de caractere şi `Boolean` = logic.

Ce notează ultimele trei tipuri? Prin `x:Char` înțelegem că `x` este un caracter, adică o literă (mică sau mare), o cifră sau un simbol special: +,[,),‘,$,#,@,:,? etc.. Prin `s: String` vom înțelege că `s` este o înşiruire de astfel de caractere, de pildă `s = a0+Z# $M`. Pentru a elimina unele ambiguități, s-a convenit să se noteze caracterele şi şirurile de caractere între apostrofuri: deci, de exemplu: `x = 'a', s = 'a0+Z# $M'`.

Prin `p: Boolean` vom înțelege că `p este` o variabilă logică, în sensul că ea poate avea doar două valori: *adevărat* (`true`) şi *fals* (`false`). Astfel, `true` şi `false` sunt constante de tipul `Boolean` şi, printr-o atribuire de forma `p:=true`, înțelegem că `p` devine adevărată, iar prin `q:=false` înțelegem că `q` devine falsă, unde `p:boolean` şi `q:boolean` (`p` şi `q` sunt de tip *boolean*).

E de la sine înțeles că aceste tipuri de date nu sunt singurele de care avem nevoie în soluționarea unei probleme cu ajutorul calculatorului. Astfel, în ordonarea crescătoare a elevilor va trebui să dispunem de acea variabilă `n` care să reprezinte numărul de elevi. În cu totul alt mod va trebui să reținem înălțimile elevilor. Astfel, fiecare înălțime este un număr real, iar ansamblul acestor înălțimi este ceea ce se numeşte un **vector** (**şir**) de numere reale. Dacă ar trebui să memorăm şi alte date despre fiecare elev în parte, atunci fiecare elev ar fi o **structură** (**înregistrare**, **articol**) mai complexă, iar mulțimea elevilor ar fi un vector de astfel de structuri.

Deci, aşa cum programele mari se realizează din îmbinarea armonioasă a procedurilor şi structurilor elementare, aşa şi tipurile de date se pot forma, pornind de la cele simple, menționate mai sus sau de la altele mai complexe, definite deja.

6. Exerciții recapitulative

1. Care este valoarea variabilei `p` după efectuarea următoarelor operații ?

a) `p:=1; i:=2; while i<7 do i:=i+1; p:=p*i;`

b) `p:=1; i:=7; while i>1 do begin p:=p*i; i:=i-1 end;`

2. Să se calculeze suma: S = 1-1·3+1·3·5- ... ±1·3·5·...·(2n-1).

3. Ce este greşit în secvența de instrucțiuni de mai jos ?

```
begin
    if n<5 then x:=while
    else i:=i+1;
    while p<4 do k:=k+1
end;
```

4. Cum puteți exprima, folosind doar simbolurile < şi >, că `x` este în întervalul deschis `(a,b)`. Puteți folosi şi cuvîntul `and` ("şi").

5. Ce valoare are variabila *p*, în urma secvenței de instrucțiuni de mai jos:

```
p:=2;
if 2<p then p:=2*p
else begin i:=1;
          while i<3 do begin p:=p+i; i:=i+1 end
     end;
```

6. Cîte instrucțiuni şi subinstrucțiuni sunt mai jos ? Încadrați-le !

```
x:=2; i:=x-5;
while p<x do
      begin
        h:=5;
        if x<h then x:=x+x
        else x:=x*3
      end;
p:=8;
```

7. Fie un şir de *n* numere a_1, a_2, ..., a_n. Să se calculeze suma:

$$S = \frac{a_1}{1} + \frac{a_2}{2} + \cdots + \frac{a_n}{n} .$$

8. Fie *a*, *b* şi *c* lungimile laturilor unui triunghi. Cum exprimați faptul că triunghiul este echilateral. Folosiți "=" şi "`and`".

9. Fie $a_1, a_2, \ldots, a_n$ n numere reale. Calculați suma: $S=a_1-a_2+a_3-\ldots a_n$.

10. Ce calculează urmatoarea secvență de instrucțiuni:

```
S:=0; i:=1; while i<n+1 do begin S:=S+2*i; i:=i+1end;
```

11. Ce este greşit în procedura de mai jos ?

```
procedure CalculSofisticat1(a,b,c: Integer);
begin
     while a<b then
           begin
             a:=a+1;
             if c>a then c:=a-b else c:=c+b;
             b:=2*b-c
           end;
```

12. Ce este greşit în procedura de mai jos ?

```
procedure CalculSofisticat2(a,b,c: Integer);
then
    if a<3
       begin
         b:=2; c:=a-b
         3:=5;
       end
end;
```

13. Ce este greşit în secvența de instrucțiuni de mai jos ?

```
a:=m-n;
i=1;
```

```
while i<4 do i:=i+1; else -1
```

14. Scrieți o procedură care să calculeze perimetrul unui dreptunghi cu lungimea `lungime` şi lățimea `latime`.

15. Scrieți o procedură care să determine minimul a trei numere `a`, `b` şi `c`. Veți folosi comparații multiple (cu simbolul <) şi instrucțiunea *if...then...else...* de cîte ori aveți nevoie.

16. Scrieți o procedură care să determine media aritmetică a două numere `a` şi `b`.

17. Care este exprimarea corectă în informatică pentru expresia de mai jos ?

$$\frac{x - y + a\ b - y}{1 - \frac{a - x}{y^2}}$$.

18. Scrieți o procedură care să calculeze aria unui triunghi în funcție de lungimile laturilor sale `a`, `b` şi `c`. Mai intîi veți determina semiperimetrul. În loc de *radical din x* scrieți `sqrt(x)`.

19. Scrieți o procedură care să determine maximul a doua numere `a` şi `b`. Veți folosi comparații (<) şi instrucțiunea `if...then...else...` .

20. Scrieți o procedură care să determine media geometrică a doua numere `a` şi `b`.

21. Scrieți o procedură care să calculeze aria unui pătrat de latură `l`.

22. Scrieți o procedură care să determine media aritmetică a `n` numere întregi.

23. Scrieți o procedură care să determine maximul dintr-o mulțime de n numere.

24. Scrieți o procedură care să calculeze, pt. fiecare $n > 0$, natural, produsul $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, notat prin `n!` şi numit *factorialul* lui `n` (`n` factorial).

25. Scrieți o procedură care să numere cîte fete şi cîți băieți sunt într-un grup de `n` elevi.

Lecția 2
ELEMENTELE DE BAZĂ ALE LIMBAJULUI

1. Identificatori. Constante

Identificatori

Un **identificator** este numele unui element dintr-un program, fie el o variabilă, un tip, o procedură sau chiar programul însuşi. Identificatorul este reprezentat de o succesiune de litere (mici sau mari), cifre şi simbolul "_". Această succesiune începe neapărat cu o literă sau cu simbolul "_".

Exemple:

- `a,b,c,x1,x2,delta` ar putea identifica, respectiv, coeficienții reali din ecuația de gradul 2, soluțiile şi discriminantul acesteia;
- `cmmdc,cmmmc` ar putea reprezenta cel mai mare divizor comun, respectiv cel mai mic multiplu comun a două numere întregi;
- `n,p,i` ar putea identifica numele, prenumele şi respectiv inițiala tatălui unei persoane, dar aceleaşi elemente ar putea fi notate, mai sugestiv, prin `nume, pren, init_tata`;
- `v,t` ar putea identifica viteza şi timpul de deplasare a unui vehicul, dar aceleaşi elemente ar putea fi notate, mai sugestiv, prin `viteza` şi `timpul`;
- `x2_ab,_,a1___xc1XC` sunt, de asemenea identificatori corecți, însă nu sugerează nimic...

Contraexemple. Erori frecvente:

- De multe ori se uită că nu se pot folosi decât litere din alfabetul latin şi se notează, de pildă, discriminantul ecuației de gradul 2 prin Δ, în loc de `delta` sau, simplu, `d`.
- Nu se pot scrie indici de genul x_1, cum am făcut până acum, ci vom scrie, de exemplu, `x1` pentru a identifica prima soluție a ecuației de gradul 2.
- Nu este voie să se folosească spații sau puncte în interiorul unui identificator; exprimări de genul `c.m.m.d.c.` sau `init tata` sunt greşite.
- Un identificator nu poate începe cu o cifră (`23ab`) şi nici nu poate conține simboluri speciale, ca în: `a$, a%, init-tata, nume&prenume` etc..

Constante

O **constantă** este o dată (o informație) a cărei valoare nu poate fi modificată pe parcursul rulării unui program. O constantă se declară în felul următor:

```
const identif_constantă=valoarea_const;
```

Putem declara şi mai multe constante, astfel:

```
const id_c_1 = val_ct_!;
      id_c_2 = val_ct_2;
       ..................
```

Exemple:

- Dacă va trebui să facem un program cu calcule despre cercuri, vom folosi constanta π, adică valoarea ei, cu cât mai multe zecimale, pentru o precizie a calculelor cât mai bună. Scrierea acestor zecimale de fiecare dată când apelăm la π ar fi obositoare, de aceea

declarăm constanta `pi` şi vom folosi `pi`, în loc de 3,1415926: `const pi=3.1415926;`. Observăm că am folosit *punctul zecimal*, în loc de virgula zecimală, în exprimarea numărului real π; aşa se procedează în programare, după sistemul de notare anglo-saxon.

- Poate că odată va trebui să scriem un program pentru admiterea la un liceu a unor elevi, liceu care are un anumit nume constant; de aceea vom declara:
`const NL='Liceul Teoretic "George Bacovia" Bacău';`. E mai simplu de scris `NL` în loc de numele liceului !

Dacă facem un joc pe calculator, vom folosi nişte taste ce vor controla, de pildă, deplasarea unei piese. De exemplu: `const stinga='o'; dreapta='p'; sus='q'; jos='a';` şi chiar dacă uităm ce taste am definit, e mult mai uşor să le referim prin identificatorii constantelor `stinga, dreapta, sus` şi `jos`.

Contraexemple. Erori frecvente:
De multe ori se scrie greşit o declarație de constantă, după cum urmează:

- `const pi;` Se uită să se precizeze valoarea lui π.
- `const pi:=3.1515926;` Se foloseşte "**:=**" (specific atribuirii) în loc de "**=**".
- `const π=3.1415926;` Se greşeşte la identificatorul de constantă.

2. Variabilă. Tip de dată

În lecția precedentă am definit noțiunea de tip de dată. Acum vom reveni pentru a prezenta operațiile ce pot fi realizate cu datele (constante sau variabile) de diferite tipuri.

Prin **variabilă** se înțelege o informație (o dată) a cărei valoare *se poate* modifica pe parcursul execuției unui program. Mulțimea de valori pe care o poate lua acea variabilă reprezintă **tipul** acelei variabile. O variabilă se declară astfel: **`var`** `identif_var : identif_de_tip;`.

Când avem mai multe variabile de acelaşi tip, putem scrie: **`var`** `v_1, v_2, v_3: tip;`, iar când avem şi tipuri mai multe, vom scrie ceva de genul:

```
var v_1, v_2, v_3: tip_1;
    w_1, w_2, w_3: tip_2;
    ......................
```

3. Tipul `Integer`

Acesta corespunde mulțimii Z a numerelor întregi.

Operațiile executabile cu numere întregi sunt:

- +, -, * (înmulțirea), iar împărțirea este cu cât şi rest: `x div y` notează câtul împărțirii lui `x` la `y`, iar `x mod y` notează restul împărțirii lui `x` la `y`.
- De asemenea, putem vorbi despre modulul lui `x`, notat prin `Abs(x)`, precum şi de pătratul lui `x`, adică x^2, notat prin `Sqr(x)`. Radicalul lui `x` este un număr real, de aceea această operație, ca şi împărțirea propriu-zisă, vor fi prezentate la tipul numerelor reale.
- Funcția `Odd(x)` ne indică paritatea unui număr real `x`. `Odd(x)=True` dacă `x` este impar, respectiv `False`, când numărul este par.

Dacă datele de un anumit tip se pot pune într-o anumită ordine, deci se pot număra, spunem că tipul este **ordinal** (**scalar**). Orice tip scalar admite operațiile: `Ord` (numărul de ordine), `Pred` şi `Succ` (predecesorul şi succesorul unui element).

Numerele întregi se pot număra, astfel încât `Integer` este un tip scalar. Sunt considerat doar numerele întregi situate între `-MaxInt-1` şi `MaxInt`, unde `MaxInt` este o constantă predefinită, reprezentând cel mai mare număr întreg cu care lucrează respectiva implementare Pascal. În cazul lui Turbo-Pascal 7.0, această valoare este `32767`. (Există şi alte tipuri întregi pentru numere mai mari în valoare absolută).
Avem: `Ord(x)` = numărul de ordine al lui `x` în lista numerelor întregi, `Ord(x)=x`. De asemenea, `Pred(x)=x-1` şi `Succ(x)=x+1`.
Exemple:

- Să considerăm declarațiile: `var a,b,m,n,i: Integer;`

În urma secvenței de atribuiri: `a:=2; b:=3+a; m:=b div 2; i:=1; n:=Sqr(2+i); i:=Pred(i);` vom avea: `a=2, b=5, m=2, n=9, i=0`.

- Dacă vrem să determinăm cel mai mare divizor comun şi cel mai mic multiplu comun a două numere întregi, vom declara ceva de genul: `var a, b, cmmdc, cmmmc: Integer;`. Cel mai mic multiplu comun se calculează împărțind produsul `ab` la `cmmdc`: `cmmmc:=a*b div cmmdc`.
- Faptul că un număr `n` este par, se poate exprima şi prin `n mod 2 = 0`.
- Faptul că `n` este divizibil prin `m` se exprimă prin: `n mod m = 0`.
- Ultima cifră a unui număr întreg `n` este `n mod 10`, iar pentru a afla celelalte cifre ale numărului, va trebui să facem o împărțire repetată prin `10` (`n:=n div 10`). Apoi se consideră din nou ultima cifră şi se repetă împărțirea.

Utilizare. Datele de tip `Integer` vor fi folosite în probleme de matematică cu numere întregi, dar şi în programe care lucrează cu ecranul, deoarece acesta poate fi privit ca un plan în care coordonatele sunt numere întregi, nu reale. De asemenea, vom folosi numere întregi când vom avea nevoie să contorizăm sau să indexăm ceva.

4. Tipul `Real`

Operații. În cazul mulțimii numerelor reale, avem: +,-,*, / (împărțirea cu zecimale), `Abs, Sqr, Sqrt` (radical de ordin 2), unele funcții trigonometrice (`Sin, Cos` etc.), precum şi alte funcții matematice pe care nu le prezentăm aici. Există trei funcții deosebite mai importante:

- `Round(x)` ne dă numărul `x` rotunjit la cel mai apropiat întreg;
- `Trunc(x)` ne dă numărul `x` trunchiat la număr întreg;
- `Int(x)` ne dă partea întreagă a numărului `x` văzută ca un număr real.

Astfel, `Round(3.4)=3, Round(-3.4)=-3, Round(3.6)=4, Round(-3.6)=-4, Trunc(3.4)=3, Trunc(-3.4)=-3, Trunc(3.6)=3, Trunc(-3.6)=-3, Int(3.4)=3.0, Int(-3.4)=-4, Int(3.6)=3.0, Int(-3.6)=-4`.

Exemple:

- Fie declarațiile: `var a,b,x,y,z:Real;m,n:Integer;` În urma secvenței de atribuiri:`a:=2;b:=3;x:=-b/a;y:=Sqr(x);z:=Abs(x);m:=Trunc(3-x); n:=Round(z*2+0.2)`, vom avea: `a=2, b=3, x=-1.5,y=2.25, z=1.5, m=4, n=3`.
- Pentru a exprima că un număr real `x` este şi întreg se va folosi funcția `Int`: `x=Int(x)`.

Utilizare. Numerele reale vor fi folosite în orice gen de probleme cu calcule ştiințifice.

5. Tipul `Char`

Acest nume identifică mulțimea caracterelor, adică a tuturor cifrelor, a literelor mari şi mici, precum şi a caracterului spațiu sau a simbolurilor speciale: `+,-,$,(,;,@` etc.. Acestea

trebuie să fie cuprinse între apostrofuri. Un caz special îl constituie apostroful însuşi, care trebuie dublat: ‘’’’. Tipul `Char` este scalar, deoarece caracterele se pot număra. De fapt, există o listă a lor, numită `ASCII`, conținând fiecare caracter şi numărul său de ordine.

0.....	65.....A
1.....	66.....B
.....	.....
32.....(spațiu)	97.....a
33 .....!	98.....b
.....	.....
.....	255 ....

Deci `Ord('A')=65`, `Ord('b')=98`. Avem şi funcțiile succesor şi predecesor: `Succ('A')='B'`, `Pred('b')='a'`. Există şi o funcție inversă funcției `Ord`, numită `Chr`, care returnează caracterul având un anumit cod `ASCII`: `Chr(65)='A'`.

Exemple:
Fie declarațiile: `var a,b,c: Char; m: Integer;` În urma secvenței de atribuiri: `a:='b';b:=Succ(a);c:=Succ('D'); m:=Ord(b);` vom avea valorile: `a='b'; b='b'; c='E'; m=98`.

Utilizare. Tipul caracter va fi folosit mai ales în programele ce folosesc tastatura.

6. Tipul `Boolean`

Este vorba despre tipul logic, adică despre mulțimea celor două valori de adevăr {`False, True`}, studiată de matematicianul englez *G. Boole*. Ordinea în mulțime nu este întâmplătoare, deoarece avem şi aici funcția `Ord`: `Ord(False)=0`, `Ord(True)=1`. De asemenea, tipul `Boolean`, fiind un tip ordinal, avem şi: `Succ(False)=True`, `Pred(True)=False`. În fine, acestea sunt mai puțin importante, ceea ce ne interesează la variabilele de acest tip sunt

Operațiile specifice:
- conjuncția logică: `x and y;`
- disjuncția logică: `x or y;`
- negația logică: `not x`.

Tabelele lor de adevăr sunt următoarele:

not	False	True
	True	False

and	False	True
False	False	False
True	False	True

or	False	True
False	False	True
True	True	True

Exemple:
- Fie declarațiile: `var cond1, cond2, a: Boolean; s: Char; m: Integer;` După atribuirile: `cond1:=2=3; cond2:=False; a:=not(cond2) or cond1;` vom avea: `cond1=False` (căci $2 \neq 3$), `cond2=False`, iar `a=True`.
- Dacă avem şi: `s:='a';a:=s='b';m:=ord(a)`,atunci `s='a',a=False, m=0`.
- Dacă vrem să precizăm că un triunghi este echilateral, putem folosi o variabilă booleană `echilateral`, care se poziționează pe `true` atunci şi numai atunci când cele trei laturi au aceeaşi lungime: `echilateral := (a=b) and (b=c) and (c=a)`. Fireşte, din `(a=b)`

şi `(b=c)` rezultă `(c=a)`, deci putem scrie, mai simplu: `echilateral:=(a=b) and (b=c)`.

- Dacă `prim` este o variabilă booleană, care indică, de pildă, că un număr este prim, atunci, pentru a preciza că variabila are valoarea adevărat, putem scrie fie `prim=True`, fie chiar numai `prim`. De asemenea, putem scrie `not prim` în loc de `prim=False`.

Utilizare. Variabilele logice sunt foarte utile în programare, atunci când trebuie să exprimăm diferite condiții ce trebuiesc verificate. În primul capitol am prezentat asfel de algoritmi.

7. Tipul `String`

Vom considera acest tip de date ca fiind predefinit, deci tip de bază, deoarece el este indispensabil în foarte multe probleme. Un şir de caractere reprezintă o succesiune de caractere cuprinse între apostrofuri.
Operații asupra şirurilor de caractere sunt multe, însă vom prezenta acum doar *concatenarea* (*alipirea*) şirurilor, notată prin "+":

Exemple:

- Dacă avem declarațiile: `var nume, prenume, nume_total: String;` în urma atribuirilor: `nume:='George'; prenume:='Bacovia'; nume_total := nume + prenume;` vom avea `nume_total='GeorgeBacovia'`, iar pentru a insera un spațiu, ar fi trebuit scris: `nume_total:=nume+' '+prenume`.
- Çirul vid este `''`, iar un şir cu un singur caracter poate fi considerat ca fiind o dată de tip `Char`.

Utilizare. `String` este un tip foarte utilizat în programare: pentru a memora diferite denumiri, precum şi pentru a prelucra texte etc..

8. Operații comune tuturor tipurilor predefinite

- Cu datele de orice tip predefinit se pot face operații de comparare (dacă valorile care se compară sunt de acelaşi tip, sau compatibile). Astfel, avem: `<, <=, >, >=, =, <>`, în care `<=` înseamnă ≤, `>=` înseamnă ≥, `<>` înseamnă ≠.

Numerele (întregi sau reale) se compară obişnuit, caracterele după lista codurilor `ASCII`, iar şirurile de caractere se compară alfabetic. Astfel avem: `'mama' < 'tata'`, `'ANA'<'ana'`, `'Ana-Maria' > 'Ana-Ioana'` şi `'Dan'<'Daniel'`. În cazul valorilor logice, avem `False<True` şi bineînțeles `False ≠ True`.

- Operația de atribuire este comună tuturor acestor tipuri date şi ea a fost prezentată în capitolul 1.

Observație. Când se lucrează cu expresii matematice, acestea se scriu liniarizat; de exemplu, soluțiile ecuației de gradul 2 sunt:
`x1:=(-b-Sqrt(Sqr(b)+4*a*c)/(2*a)` sau `x1:=(-b-Sqrt(Sqr(b)+4*a*xc)/2/a`.

9. Structura unui program în limbajul Pascal

Un program are două părți: **partea declarativă** şi cea **operativă**. În prima apare numele programului (un identificator), apoi se fac toate declarațiile elementelor ce vor fi folosite în

program: constante, variabile etc.. Partea operativă este programul propriu-zis, reprezentat de o secvență de instrucțiuni, cuprinse între cuvintele `begin` şi `end`.

Să scriem un prim program: calculul ariei unui cerc în funcție de raza sa.

```
program CalculArie;
const pi=3.1415926;
var raza, aria: Real;
begin
       raza:=2;
       aria:=pi*Sqr(raza)
end.
```

Dacă pornim în execuție acest program (în mediul Turbo-Pascal), vom observa că pâlpâie puțin ecranul, dar nu se întâmplă nimic. E şi normal. Calculatorul a atribuit valoarea `2` variabilei `raza`. A calculat `aria` (obținând o anumită valoare) şi gata. Noi nu am precizat prin nimic să ni se comunice rezultatul. De aceea va trebui să folosim o procedură de extragere a datelor, de scriere a lor pe ecran.

10. Introducerea şi extragerea datelor

Dar, chiar dacă am dispune de o asemenea procedură, rezultatul programului ar fi acelaşi la orice executare a sa. De aceea, ar fi bine să putem introduce de la tastatură raze de diferite valori; deci am avea nevoie de o procedură de citire a datelor. Aceste proceduri există în limbajul Pascal:

a. De citire a datelor (*eu scriu - calculatorul citeşte*):

- `ReadLn(`v_1`, `v_2`, ..., `v_n`)` - citeşte variabilele v_1, ..., v_n, introduse de la tastatură. Variabilele pot fi de orice tip învățat până acum, mai puțin `Boolean`. Introducerea mai multor variabile se poate face dacă acestea se separă prin spații sau `<Enter>`. La sfârşit trebuie apăsată tasta `<Enter>`.

Dacă variabilele ar fi avut alte valori înaintea apelului procedurii de citire, ele se pierd. Valorile citite ale acestor variabile rămân neschimbate până la prima atribuire sau (re)citire a lor[1].

b. De scriere a datelor (*calculatorul scrie - eu citesc*):

- `Write(`e_1`,`e_2`,...,`e_n`)` - evaluează şi apoi afişează valorile expresiilor e_1, ..., e_n. Expresiile pot fi inclusiv de tipul `Boolean`, caz în care se afişează unul din cele două cuvinte (`False` sau `True`). După afişarea valorilor unor expresii, acestea nu se pierd şi nici nu se schimbă.
- `WriteLn(`e_1`,`e_2`,...,`e_n`)` face acelaşi lucru, dar la sfârşitul întregii afişări, mută cursorul pe rândul următor.

De exemplu, pentru a citi raza vom scrie: `ReadLn(raza)`, iar, după calculul ariei, aceasta se afişează prin: `Write(aria)`. Putem scrie direct `Write(pi * Sqr(raza))`, efectul fiind acelaşi.

Exemple:

[1] Există şi o procedură *Read* de citire a datelor, care va fi prezentată la fişierele text, în capitolul 9, deoarece necesită unele explicații suplimentare.

Fie declarațiile de variabile: `var a: Integer; s: String;` Fie următoarea secvență de program: `a:=20; ReadLn(a,s); WriteLn(a*2,s+s); Write(s+'s');` în cazul introducerii valorilor `3` şi ***`alfa`***, se realizează următoarele: `a` primeşte valoarea `20`; după citire, `a` îşi pierde valoarea inițială (`20`) şi devine `3`. `s` devine '`alfa`'. Se afişează, pe acelaşi rând: `6alfaalfa`. Apoi, pe rândul următor, se afişează: `alfas`. Cursorul rămâne imediat după `s`, pe acest rând.

Erori frecvente:
Unii începători sau cunoscători ai limbajului `BASIC` cred că se poate face o citire în felul următor: `ReadLn('Dati n = ',n);` iar alții nu sesizează diferența între un şir şi valoarea identificatorului corespunzător: `ReadLn('n').`

Să rescriem programul nostru, folosind noile proceduri învățate:

```
program CalculArie;
const pi=3.1415926;
var raza, aria: Real;
begin
        ReadLn(raza);
        aria:=pi*Sqr(raza);
        WriteLn(aria)
end.
```

Pornind acest program se întâmplă următoarele:
- apare cursorul şi noi introducem o valoare, de exemplu `2`;
- se afişează foarte rapid o altă valoare, însă sub o formă ciudată: `1.25663704000035E+0001`. (??!!??) (Pentru a vedea ce s-a afişat, se va apăsa `Alt-F5`.). Este vorba despre formatul ştiințific de afişare a valorilor reale; valoarea de mai înainte se interpretează ca fiind `1.25`.....$\times 10^{1}$. Pentru a afişa o expresie pe `m` poziții, din care `n` poziții pentru partea zecimală, vom scrie: `Write(e:m:n).` De exemplu, `Write(aria:7:3)` va afişa, cu un spațiu în față: `12.566`. O poziție a fost ocupată de punctul zecimal, trei de cele trei zecimale cerute, două de partea întreagă (`12`), iar un spațiu a fost pus în față pentru a obține totalul de `7` poziții.

Cu această înlocuire, programul devine mai "bun", dar tot nu ne satisface următoarele cerințe:
- să se ceară explicit să introducem raza;
- să se spună că rezultatul pe care îl afişează este aria cercului de rază dată.

De aceea vom scrie programul de mai jos. Acesta conține un **comentariu**, adică un text cu rol explicativ, cuprins între acolade. Comentariile sunt de folos atât programatorului, cât şi unui eventual cititor al programului.

```
program CalculArie;
{ acest program calculeaza aria unui cerc in functie de raza sa }
const pi=3.1415926;
var raza, aria: Real;
begin
        WriteLn('Program de calculat aria cercului');
        WriteLn('**************************');
        Write('Dati raza = '); ReadLn(raza);
        aria:=pi*Sqr(raza);
        WriteLn('Aria este = ', aria:7:3)
end.
```

Acum lucrurile stau mai bine, însă ne mai deranjează două lucruri:
- trebuie să apăsăm `Alt-F5` pentru a vedea rezultatul programului;

- rămân pe ecran "urme" deranjante, de la execuția programelor anterioare.

Pentru a soluționa prima problemă se va folosi un apel al procedurii `ReadLn` fără parametri, înainte de `end`-ul final. Această instrucțiune va aştepta să apăsăm `Enter`, apoi va reveni în mediul Turbo-Pascal.

11. Utilizarea bibliotecii `Crt` pentru lucrul cu ecranul

Pentru a şterge ecranul înaintea începerii programului, va trebui să folosim o procedură cu numele `ClrScr` (de la *clear screen*). Scriind programul cu `ClrScr` la început şi `ReadLn` la sfârşit, observăm că o compilare afişează eroarea: *nu cunosc identificatorul* `ClrScr`. E şi firesc, acest identificator nu se află în nucleul limbajului, dar el reprezintă o procedură care afectează ecranul, iar toate procedurile care se referă la ecran s-au grupat într-o bibliotecă (*unit*), cu numele `Crt`. Dacă dorim să apelăm proceduri sau să utilizăm alte elemente din această bibliotecă, se declară `uses Crt` în cea de a doua linie a programului.

```
program CalculArie;
uses Crt;
const pi=3.1415926;
var raza, aria: Real;
begin
       ClrScr;
       WriteLn('Program de calculat aria unui cerc'); WriteLn;
       Write('Dati raza = '); ReadLn(raza);
       aria:=pi*Sqr(raza);   WriteLn('Aria   este   =   ',aria:7:3);
       ReadLn
end.
```

Acum nu ne mai deranjează nimic, însă putem începe cu mofturile ! Ce-ar fi să scriem pe un ecran albastru, cu diferite culori: despre rază cu galben, iar despre arie cu alb pe verde... Dar la mijlocul ecranului nu putem scrie ? Putem să facem scrisul clipitor ? Putem să producem un sunet în difuzor? Toate aceste întrebări au răspunsuri afirmative.

Biblioteca `Crt` ne ajută în acest sens.

- Mai întâi, trebuie precizat că ecranul este împărțit în 80 de coloane şi 25 de rânduri. Poziționarea cursorului pe coloana `c` şi linia `l` se face prin `GoToXY(c,l)`.
- Sunt disponibile 16 culori, care sunt identificate prin numele unor constante simbolice: `Black`, `Blue`, `Green`, `Magenta`, `Red`, `Brown`, `Cyan`, `DarkGray`, `LightGray`, `LightBlue`, `LightGreen`, `LightMagenta`, `LightRed`, `LightCyan`, `Yellow`, `White`, sau pot fi folosite numere întregi de la `0` la `15`.
- Efectul de "clipire" se obține adăugând valoarea `128` (constanta `Blink`) unei culori.
- Culoarea "cernelei" se alege cu `TextColor(culoare)`, a fondului cu `TextBackground(culoare)`. Un apel `ClrScr` după un apel `TextBackground(c)` va avea ca efect ştergerea şi colorarea în întregime a ecranului în culoarea `c`.
- Sunetele se pot produce astfel: `Sound(frecventa);Delay(durata);NoSound`.

Iată o variantă mai atrăgătoare a programului pentru calculul ariei cercului:

```
program CalculArieCerc;
{ programul demonstreaza posibilitatile bibliotecii CRT }
uses Crt;
const pi=3.1415926; var raza, aria: Real;
begin
     TextBackground(Blue); ClrScr;
     GoToXY(35,10); WriteLn('CALCUL ARIE CERC');
     Delay(1000); { sta o secunda degeaba }
     TextColor(Yellow); GoToXY(35,12);
```

```
        Write('Dati raza = '); ReadLn(raza);
        Sound(700); Delay(100); Sound(800); Delay(100); NoSound;
        aria:=pi*Sqr(raza);
        TextBackground(Green); TextColor(White);
        GoToXY(35,13); Delay(3000); { "gândeste" 3 secunde! }
        WriteLn('Aria este = ',aria:7:3); Sound(400); Delay(200);
        NoSound;
        { afisaj explicativ clipitor }
        TextBackground(LightCyan); { azuriu deschis }
        TextColor(1+Blink); { albastru clipitor }
        GoToXY(65,22); WriteLn('Apasa <enter> !'); ReadLn
end.
```

Mult zgomot pentru ... mai nimic !

12. Exerciții recapitulative

1. Să se determine aria şi perimetrul unui triunghi cu laturile de lungimi `a`, `b` şi `c`.
2. Un punct material se află la distanța `x0` de origine, la momentul inițial `t0`, când începe să se mişte rectiliniu uniform. Çtiind că la momentul `t` se află la distanța `x` față de origine, să se determine viteza `v` de mişcare a punctului material.
3. Să se determine aria unui trapez care are baza mică `b1`, baza mare `b2` şi înălțimea `h`.
4. Să se determine aria unui cerc în funcție de a) diametrul său; b) lungimea sa.
5. Să se determine lungimea unui cerc în funcție de aria sa.
6. Catetele unui triunghi dreptunghic au lungimile `a` şi `b`. Să se determine ipotenuza (`c`), perimetrul şi aria.
7. O carte are un număr de `nr_f` foi. Fiecare pagină a cărții are câte `nr_r` rânduri. Să se determine numărul total de rânduri din carte.
8. Viteza unui automobil este de `vit_1` km/h. Exprimați această viteză în m/s, calculând-o în variabila `vit_2`.
9. Un muncitor realizează o lucrare în `nr_zile` zile. Să se determine în câte zile realizează aceeaşi lucrare un număr de `nr_munc` muncitori.
10. Două echipe de muncitori au în componență `nr_m1`, respectiv `nr_m2` membri. Prima echipă realizează o lucrare în `nr_z1` zile. Determinați în câte zile realizează a doua echipă aceeaşi lucrare, rezultatul depunându-se în variabila `nr_z_2`.
11. Într-o magazie sunt următoarele produse: fasole, cartofi şi roşii în cantitățile `cant_fasole`, `cant_cartof`, respectiv `cant_rosii` kilograme. Un kilogram de fasole costă `pret_fasole`, un kilogram de cartofi costă `pret_cartofi`, iar un kilogram de roşii costă `pret_rosii` lei. Care este prețul total general (`pret_total`) al tuturor legumelor din magazie.
12. Într-o curte sunt `nr_gaini` găini şi `nr_pisici` pisici. Să se determine câte aripi şi câte picioare sunt în curte.

Lecția 3
INSTRUCȚIUNI DE CONTROL

1. Instrucțiunea de decizie `IF`

Necesitate

Credem că simțiți nevoia unei instrucțiuni de decizie, de pildă într-un exemplu ca acela al determinării perimetrului unui triunghi cu lungimile laturilor `a, b` şi `c`. Calculul perimetrului este simplu, dar se pune problema: întotdeauna trei numere reale pot fi laturile unui triunghi? Fireşte că nu; în primul rând ele trebuie să fie strict pozitive, în al doilea rând trebuie ca fiecare din ele să fie strict mai mic decât suma celorlalte două. Acest lucru nu poate fi evidențiat cu simpla instrucțiune de atribuire. De aceea avem nevoie de instrucțiunea `if`. Ea corespunde întru totul structurii decizionale, prezentate în primul capitol.

Sintaxă

Instrucțiunea `if` are două forme. În prima, instrucțiunea are două ramuri:

```
if cond then
   instr1
else
   instr2
```

Cea de a doua formă este fără ramura: `else instr2`:

```
if cond then
   instr1
```

În cele de mai sus, `cond` este o condiție, adică:

- fie o constantă booleană (`True` sau `False`), deşi nu are sens o asemenea utilizare;
- fie o variabilă booleană, care poate avea una din valorile `True` sau `False`;
- fie o expresie booleană, de exemplu `(e1 and (not e2)) or e3`, unde `e1`, `e2`, `e3` sunt variabile sau constante booleene;
- fie o expresie relațională, de genul `x<y, x>=y`, unde `x` şi `y` sunt variabile de acelaşi tip;
- fie o expresie mixtă, de exemplu `(x<y) and (e1 or (z>=t) and True)`.

De fapt, toate acestea sunt expresii booleene.

De asemenea, atât `instr1`, cât şi `instr2`, sunt instrucțiuni, adică:

- fie o instrucțiune vidă, adică nimic;
- fie o instrucțiune de atribuire, de exemplu `v:=e`;
- fie o instrucțiune apel de procedură, de exemplu `ReadLn(x)` sau `TextColor(Green);`
- fie o altă instrucțiune `if`, deci putem avea instrucțiuni `if` imbricate;
- fie un alt gen de instrucțiune (repetitivă, de decizie multiplă, de selecție), ce vor fi prezentate ulterior;
- fie o instrucțiune compusă.

Prin **instrucțiunea compusă** se înțelege o secvență de instrucțiuni de orice gen, încadrate de cuvintele **begin** şi **end**.

Semantică

Semantica acestei instrucțiuni de control este următoarea: se testează ce valoare de adevăr are expresia booleană `cond`. Dacă ea este `True`, atunci se execută instrucțiunea `instr1`,

după care se trece la instrucțiunea ce ar urma după întregul ansamblu `if-then-else`. Dacă are, însă, valoarea `False`, atunci se execută, în cazul în care există, instrucțiunea `instr2`, apoi se trece după ansamblu.
Aşadar, se execută fie `instr1`, fie, dacă există, `instr2`. Dacă `instr2` nu există, iar `cond=True`, atunci nu se execută nimic, iar programul continuă cu instrucțiunea de după ansamblul `if-then-else`.

Exemple
1. Faptul că `a`, `b` şi `c` pot fi lungimile laturilor unui triunghi se poate exprima în mai multe feluri, de exemplu:

```
if (a>0) and (b>0) and (c>0) then
   if (a<b+c) and (b<c+a) and (c<a+b) then
        WriteLn('Formează triunghi...')
   else WriteLn('O latură e prea mare !')
else WriteLn('Laturi negative !');
```

sau:

```
if (a>0) and (b>0) and (c>0) and (b+c>a) and (c+a>b) and (a+b>c) then
    WriteLn('Formează triunghi...')
else WriteLn('Nu formează triunghi !');
```

Faptul că *a,b* şi *c* reprezintă lungimile laturilor unui triunghi isoscel se poate preciza prin: `if (a=b) or (b=c) or (c=a) then WriteLn('Triunghi isoscel...');`

Erori frecvente
Printre cele mai întâlnite erori legate de instrucțiunea `if` avem:

- Se pune "`:=`" în loc de "`=`", ca în: `if a:=0 then WriteLn('Ecuatia nu e de gr 2')`. Niciodată nu vom avea "`:=`" într-o condiție dintr-un `if`, deoarece acest grup de simboluri reprezintă instrucțiunea de atribuire!
- Se pune "*;*" înainte de `else`, ca mai jos:

```
if x=y then WriteLn('Egalitate !') ;
else WriteLn('Diferenta !')
```

Este mare greşeală, deoarece simbolul "**;**" separă două instrucțiuni, or cuvântul else este o parte constituentă a instrucțiunii if-then-else.

- Se pune, greşit, punct şi virgulă după `then`, ceea ce este interpretat ca fiind o instrucțiune vidă, pentru `if`, urmată de o altă instrucțiune: `if cond then ; instr`
- Se pune o condiție suplimentară, inutilă, ca mai jos:

```
if x<y then
   instr1
else x>=y
   instr2
```

Evident, dacă `x` nu e mai mic decât `y`, va fi mai mare sau egal, deci nu trebuie precizat şi, de fapt, nici nu este corect sintactic!
Se foloseşte cuvântul `do` în loc de `then`, ca în: `if cond` *do* `instr1`

Programe demonstrative
Mai întâi, vom prezenta un program care rezolvă ecuația de gradul I `ax+b=0`. Apoi vom trece la rezolvarea ecuației de gradul II, $ax^2+bx+c=0$, doar pentru cazul când $a\neq0$. Propunem cititorului, cu titlu de exercițiu, să combine cele două programe astfel încât să se obțină o rezolvare completă a ecuației de gradul II.

```
program EcuatiaDeGradul_1;
var a,b,x: Real;
begin
     WriteLn('Rezolvam ec. de gradul I ax+b=0',Chr(7));
     Write('Dati a = '); ReadLn(a); Write('Dati b = '); ReadLn(b);
```

```
     if a=0 then
        if b=0 then WriteLn('Nedeterminare')
         else WriteLn('Imposibilitate')
     else begin x := -b/a; WriteLn('Solutia este: x = ',x:7:3) end;
     ReadLn
end.
```

Caracterul cu codul `ASCII 7` este neafişabil, "afişarea" sa însemnând producerea unui sunet în difuzorul calculatorului.

Pentru ecuația de gradul II, avem următorul algoritm de rezolvare:

- Este `a=0` ? Dacă da, ecuația nu mai este de gradul II, ci de gradul I.
- Dacă `a` nu este nul, atunci se calculează discriminantul ecuației, Δ. Acesta poate fi pozitiv sau `0` (caz în care se calculează, conform formulelor matematice cunoscute, cele două soluții), dar poate fi şi negativ, caz în care se va afişa un mesaj de avertizare.

```
program EcuatiaDeGradul2;
var a,b,c,x1,x2,delta: Real;
begin
     WriteLn('Rezolvam ecuatia de gradul II');
     WriteLn('Dati coeficientii !'); ReadLn(a,b,c);
     if a=0 then WriteLn('Ecuatia e de gradul I')
     else
        begin
          delta:=Sqr(b)-4*a*c;
          if delta>=0 then
             begin
               x1:=(-b-Sqrt(delta))/(2*a); x2:=(-b+Sqrt(delta))/(2*a);
               WriteLn('Solutiile sunt:');
               WriteLn(x1:8:4); WriteLn(x2:8:4)
             end
          else WriteLn('Nu sunt solutii reale !')
        end;
     ReadLn
end.
```

Exerciții

1. Să se determine dacă trei numere reale `a`, `b` şi `c` pot reprezenta lungimile laturilor unui triunghi. Dacă da, să se determine perimetrul său, altfel să se producă un sunet în difuzor.

2. Să se determine de ce tip (isoscel, echilateral, dreptunghic) este un triunghi cu laturile `a`, `b` şi `c`.

3. Să se determine maximul *şi* minimul a două numere întregi `a` şi `b`.

4. Să se citească două nume de persoane `nume1` şi `nume2`. Să se afişeze cele două nume în ordine alfabetică.

5. Să se determine dacă un număr `a` este divizibil cu toate numerele `b1`,`b2`,`b3`.

6. Să se scrie un program care să citească două numere reale `a` şi `b`. Apoi să pună utilizatorului o întrebare: *Ce doriți să calculăm ? Media aritmetică (1) sau geometrică (2)?*. Dacă se va răspunde prin `1`, se va calcula şi afişa media aritmetică, iar pentru `2` media geometrică (numai dacă numerele sunt pozitive !, iar de nu, se va afişa '*eroare !*'). Dacă nu se răspunde prin `1` sau `2` se va produce un sunet în difuzor.

7. Să se scrie un program care să citească trei numere reale `a`, `b` şi `c`, apoi să pună o întrebare de genul: *Ce doriți să calculăm ? Aria sau perimetrul ?*. Dacă se va răspunde prin '*aria*' atunci se va calcula şi afişa aria, altfel perimetrul. Ce se va întîmpla dacă se va răspunde prin '*arie*' ?

8. Să se scrie un program care să citească un număr întreg, iar dacă acest număr este `1` să afişeze *luni*, dacă este `2` să afişeze *marți*, ... `7` - *duminică*, iar dacă nu este cuprins între `1` şi `7` să afişeze cuvîntul *eroare*.
9. Să se scrie un program care să calculeze aria unui cerc în funcție de diametru, dacă răspunsul la întrebarea: '*Aria în funcție de diametru (1) sau lungime (2)?*' este fie caracterul '`1`', fie caracterul '`D`', sau să calculeze aria cercului în funcție de lungime, dacă răspunsul este '`2`' sau '`L`', iar dacă răspunsul este diferit de aceste patru posibilități să se producă un sunet.
10. Venitul unei societăți comerciale este de `venit` milioane lei, iar cheltuielile sunt de `chelt` lei. Să se precizeze dacă profitul societății este mai mare sau nu decât o valoare constantă `profit_minim = 50` (milioane lei).
11. Să se precizeze dacă un triunghi cu lungimile celor trei laturi `a`, `b` şi `c` este sau nu isoscel. În caz afirmativ, să se producă un sunet şi să se afişeze *Tra-la-la* în mijlocul ecranului, altfel să se rezolve ecuația `ax+b=0`, unde `a` şi `b` sunt lungimile primelor două laturi ale triunghiului.

2. Instrucțiunea repetitivă `WHILE`

Necesitate

De ce o instrucțiune repetitivă ? Evident, repetiția unei secvențe este necesară, iar acest lucru s-ar putea realiza cu o instrucțiune de salt. Or, tocmai instrucțiunea de salt este marele duşman al programării structurate, motiv pentru care avem nevoie de instrucțiuni repetitive. O astfel de instrucțiune, familiară cititorului, este instrucțiunea `while`. Ea corespunde întru totul structurii iterative din programarea structurată (vezi lecția 1), având condiția de test la începutul ciclului. Prin urmare, e posibil ca ciclul să nu se execute deloc, dacă din start condiția nu este îndeplinită.

Sintaxă

Această instrucțiune are forma următoare:

```
while condiție do
      instr
```

Ca şi la instrucțiunea `if`, şi aici `cond` este o condiție oarecare, iar `instr` o instrucțiune oarecare, inclusiv o instrucțiune `if` sau o altă instrucțiune `while`. De cele mai multe ori avem de a face cu execuția repetată a mai multor instrucțiuni, deci se va folosi o instrucțiune compusă, ceva de genul:

```
while cond do
      begin
         instr1 ;
         instr2 ;
         ........
         instrn
      end;
```

Semantică

O structură de genul următor:

```
while cond do
        instr;
instr_urm
```

trebuie interpretată astfel:

- se evaluează condiția `cond`;

- dacă este adevărată, atunci se execută `instr`, apoi se revine la evaluarea condiției (care între timp s-ar fi putut modifica) şi aşa mai departe;
- dacă la un moment dat condiția este falsă, atunci se termină ciclul şi se trece la instrucțiunea succesoare ciclului, adică la `instr_urm`;
- dacă de la bun început `cond` era falsă, se trecea la `instr_urm`.

Pe scurt, cât timp condiția `cond` este adevărată, se execută instrucțiunea `instr`. După ciclu, se execută instrucțiunea `instr_urm`.

Exemple

De exemplu, să facem suma `S=1+2+3+...+n`. Plecăm cu `S=0` şi adunăm 1, apoi 2, apoi 3 ş.a.m.d. până adunăm şi pe `n`. De fapt, adunăm `i`, unde `i` porneşte cu 1 şi ajunge să fie `n`. La fiecare pas, el creşte cu o unitate. Programul complet este:

```
program CalculSuma;
var i,S,n: Integer;
begin
       WriteLn('Calculam 1+2+...+n'); WriteLn('Dati limita n !');
       ReadLn(n);
        S:=0; i:=1;
        while i<=n do begin S:=S+i; i:=i+1 end;
        WriteLn('Suma este = ',S);
        ReadLn
end.
```

Un alt exemplu va folosi biblioteca `Crt`, despre care am vorbit şi în capitolul 2.

Pentru a lucra cu tastatura, `Crt` ne pune la dispoziție două funcții speciale:

- o funcție booleană `KeyPressed`, fără argument, care returnează `True` dacă "tocmai" s-a apăsat o tastă;
- o funcție fără argument `ReadKey` care returnează un caracter, anume caracterul corespunzător tastei apăsate (pentru litere mari, de exemplu, se va apăsa şi `Shift`).

Astfel, dacă vrem să facem un program care să afişeze codurile tastelor apăsate, iar altfel (dacă nu se apasă nimic) să se afişeze un mesaj corespunzător, va trebui să scriem un program după modelul următor, în care caracterul cu codul `ASCII 27` este, de fapt, tasta `Escape`:

```
program TestTaste1;
uses Crt;
var tasta: Char;
begin
     while tasta<>Chr(27) do
          if KeyPressed then
             begin
               tasta:=ReadKey;
               WriteLn(Ord(tasta)); Delay(700)
             end
         else WriteLn('Nimic apasat !')
end.
```

O simplă modificare în program va afişa codul tastei apăsate, aşteptând şi apăsarea ei:

```
program TestTaste2;
uses Crt; var tasta: Char;
begin
     while tasta<>Chr(27) do
        begin tasta:=ReadKey; WriteLn(Ord(tasta)) end
end.
```

Erori frecvente

- Ca şi în cazul lui `if`, putem să întâlnim la începători, ceva de genul:

```
while a:=b do
          instr;
```

lucru complet greşit !

- Unii vor să execute mai multe instrucțiuni într-un ciclu, însă uită să le cuprindă între `begin` şi `end`:

greşit:
```
while cond do
        instr1;
        instr2;
        .......
        instrn
```

corect:
```
while cond do
     begin
        instr1;
        .......
        instrn
    end
```

- Alții fac greşeli semantice, ducând la cicluri infinite, ca de pildă:

```
tasta:= ReadKey;
while tasta<>'#' do
       begin
         { aici e o secvență de instrucțiuni care nu afectează variabila tasta }
       end;
```

- O interesantă eroare întâlnită la unii (care greşesc, dealtfel, în acest mod şi la instrucțiunea `if`) este scrierea unui "**;**" după cuvântul `do`. Turbo-Pascal va interpreta ca un ciclu `while` cu o instrucțiune vidă, urmată de altă instrucțiune, eventual compusă !

```
while cond do ;
      instr
```

Fireşte, compilatorul nu va semnala eroare sintactică, aşa încât un asemenea "punct şi virgulă" poate da mare bătaie de cap !
La fel şi în cazul unui `if` simplu (fără `else`) !

```
if cond then ;
   instr
```

- În fine, unii începători încurcă instrucțiunile `if` şi `while` între ele şi scriu, de pildă:

```
while cond then instr
```

Program demonstrativ

Să scriem un program care determină c.m.m.d.c. şi c.m.m.m.c. a două numere întregi `a` şi `b`. Vom folosi algoritmul lui Euclid în varianta cu scăderi repetate: cît timp numerele sînt diferite între ele, vom scădea pe cel mic din cel mare. Atunci când numerele devin egale, fiecare din ele este cel mai mare divizor comun. Cel mai mic multiplu comun se determină împărțind produsul celor două numere la cel mai mare divizor comun al lor. Trebuie salvate variabilele `a` şi `b`, reprezentând cele două numere, în alte două variabile (`u` şi `v`), deoarece, în urma scăderilor repetate, `a` şi `b` îşi pierd valorile inițiale. De asemenea, scăderile nu au sens atunci când unul din numere ar fi zero. În acest caz, cel mai mare divizor comun este cel de al doilea număr. Dacă, însă, acest de al doilea număr este tot zero, nu există un cel mai mare divizor comun al celor două numere. De asemenea, când unul din numere este zero, cel mai mic multiplu comun este 0. Programul este următorul:

```
program CmmdcSiCmmmc;
var a, b, u, v, cmmdc, cmmmc: Integer;
begin
     Write('Dati a = '); ReadLn(a); Write('Dati b = '); ReadLn(b);
     if a*b=0 then { adica a=0 sau b=0 }
        begin
          if a+b<>0 then { macar un numar este diferit de zero }
             WriteLn('c.m.m.d.c. = ',a+b)
            { adica a (cind b=0) sau b (cind a=0) }
```

```
            else WriteLn('nu exista c.m.m.d.c.');
            WriteLn('c.m.m.m.c. = 0')
          end
       else
          begin
            u:=a; v:=b;
            while a<>b do if a>b then a:=a-b else b:=b-a;
            cmmdc:=a; cmmmc:=u*v div cmmdc;
            WriteLn('c.m.m.d.c. = ',cmmdc);
             WriteLn('c.m.m.m.c. = ',cmmmc)
          end;
       ReadLn
end.
```

Exerciții

1. Să se calculeze factorialul unui număr întreg `n`, definit prin:

`n! = 1×2×...×n.`

2. Să se simuleze mişcarea unei bile pe o masă de biliard, fără frecare.

3. Să se deplaseze acționând patru taste (sus, jos, stînga, dreapta) o literă de pe ecran.

4. Să se realizeze următorul joc: o bilă se mişcă pe ecran ca pe o masă de biliard (vezi problema 2). O paletă de lungime 9 (caractere), situată pe ultima linie a ecranului, se deplasează stânga-dreapta cu ajutorul a două taste (vezi problema 3). Când bila loveşte paleta, în difuzor se aude un sunet mai lung, iar când ea loveşte pereții laterali, un alt sunet, mai scurt. Jocul se opreşte când se apasă o anumită tastă de stop.

6. Calculul radicalului. Să se calculeze $\sqrt{x}$ unde $x \in R$. Se va folosi *metoda lui Newton*. Această metodă constă în următoarele:

- pornind cu $a_0 = 1$, se generează, recursiv, următorul şir de numere reale:

$$a_n = (a_{n-1} + x / a_{n-1}) / 2.$$

- când diferență dintre a_n şi a_{n-1} este foarte mică (mai mică decât o limită ε dată), procesul de generare a lui a_n încetează;
- la sfârşit, a_n reprezintă rădăcina pătrată a lui *x*.

7. De la tastatură se introduce o listă de numere întregi. Se cere să se afişeze valoarea maximă depistată în listă. Dimensiunea listei este precizată înainte de a fi introduse elementele ce o compun.

8. Să se scrie un program care să deseneze un dreptunghi umplut cu o anumită culoare (o cutie). Dreptunghiul se va specifica prin coordonatele colțurilor stânga-sus şi dreapta-jos, care, alături de culoare, vor fi introduse de la tastatură.

9. Să se scrie un program care să deseneze pe ecran dreptunghiuri de dimensiuni aleatoare, în poziții aleatoare şi de culori aleatoare pe ecran, pînă se acționează o tastă.

Rezolvare:

Mai întâi trebuie precizat că în limbajul Turbo-Pascal există posibilitatea de a **genera numere la întâmplare**:

- un număr real aleator cuprins în intervalul $[0,1]$ se poate obține prin apelul funcției `Random`, fără argumente;
- un număr întreg aleator, între `0` şi `n-1` se poate obține cu un apel `Random(n)`.

De fapt, la fiecare apel al lui `Random` (într-unul din cele două moduri) calculatorul selectează câte un număr dintr-o secvență de numere aleatoare, însă, pentru ca numărul de start al acestei secvențe să fie mereu altul, se va apela, mai întâi, procedura `Randomize`.

Pentru a genera numere întregi aleatoare între două limite `m` şi `n` vom scrie: `x:=m+Random(n-m).`

O altă problemă ce apare în acest program este următoarea: dacă se generează coordonatele colțului stânga sus mai mari decât cele ale colțului opus ? În acest caz, va trebui să facem interschimbări de coordonate.

Astfel, avem de a face cu **problema interschimbării a două variabile** de acelaşi tip, fie ele `x` şi `y`. Pentru aceasta, nu putem scrie `x:=y; y:=x,` deoarece, evident, în momentul în care lui `x` i-am dat valoarea lui `y`, `x` şi-a pierdut fosta valoare, pe care trebuie să i-o dea lui `y`! Prin urmare, va trebui să păstrăm, la un moment dat, valoarea lui `x` în două locuri, deci e nevoie de o variabilă suplimentară `aux`, de acelaşi tip cu `x` şi `y`.
Vom scrie, aşadar: `aux:=x; x:=y; y:=aux`.
Există, însă, posibilitatea de a schimba pe `x` cu `y`, procedând astfel (verificați !):
`x:=x+y; y:=x-y; y:=x-y`. Frumos şi simplu, nu ?

Cu aceste precizări, programul nostru se scrie astfel:

```
program DeseneazaCutiiAleatoare;
uses Crt;
var x1,y1,x2,y2,x,y,aux: Integer;
begin
  TextColor(White); TextBackground(Black); ClrScr; Randomize;
  while not KeyPressed do
   begin
    Delay(100); x1:=Random(80)+1; x2:=Random(80)+1;
    if x1>x2 then
       begin { schimb pe x1 cu x2 } aux:=x1; x1:=x2; x2:=aux end;
    y1:=Random(25)+1; y2:=Random(25)+1;
    if y1>y2 then
       begin { schimb pe y1 cu y2 } aux:=y1; y1:=y2; y2:=aux end;
    TextBackGround(Random(16)); x:=x1;
    while x<=x2 do
       begin
         y:=y1;
         while y<=y2 do begin GoToXY(x,y); Write(' '); y:=y+1 end;
         x:=x+1
       end
   end;
  TextBackground(0); ClrScr
end.
```

10. Să se verifice dacă **un număr întreg este prim sau nu**.
11. Să se "inverseze" (oglindească) un număr (care nu se termină cu cifra 0). De exemplu, pentru numărul `10758` să obținem numărul `85701`.
12. Să se transforme un număr din baza 10 în baza 2.
13. Să se transforme un număr natural din baza 10 într-o bază oarecare `p`, unde `p` este un număr mai mic decât 10.

3. Instrucțiunea repetitivă cu contor FOR

Necesitate
În secțiunea precedentă (vezi problemele 8 şi 9), am reuşit să desenăm un dreptunghi colorat, cu colțul stânga sus de coordonate (`x1,y1`), iar cel dreapta jos de coordonate (`x2,y2`) folosind două instrucțiuni `while` imbricate una în cealaltă. Ambele cicluri cuprindeau câte o incrementare a unei variabile (`x`, respectiv `y`), care contoriza coloana, respectiv linia curentă de afişare. Acest lucru ar fi de dorit să se facă automat.
De asemenea, când vrem să calculăm o sumă, de pildă: `S=1+3+5+7+...+2n-1`, scriem astfel: `S:=0; i:=1; while i<=n do begin S:=S+2*i-1; i:=i+1 end`. Deci, pe

lângă actualizarea sumei, are loc şi o incrementare a contorului `i`. Acest contor pleacă de la 1 şi ajunge până la `n`.

Ar fi de dorit ca orice secvență de genul:

```
i:=val_init;
while i<=val_fin do
        begin
            instr;
            i:=i+1
        end;
```

să se scrie mai uşor, deoarece este foarte folosită. Acest lucru îl putem realiza cu instrucțiunea repetitivă cu contor, `for`.

Sintaxă

Instrucțiunea are două forme:

- Cu contorul în creştere:

```
for v:=e1 to e2 do
    instr
```

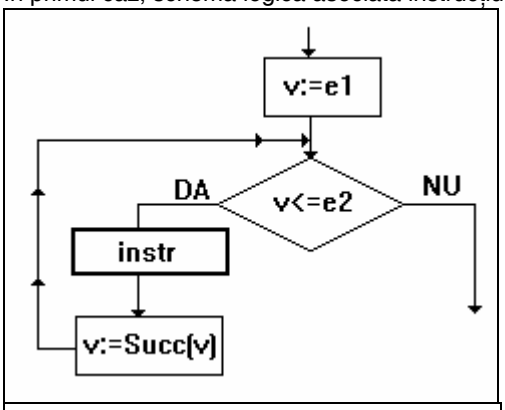
- şi cu contorul în descreştere:

```
for v:=e1 downto e2 do
    instr
```

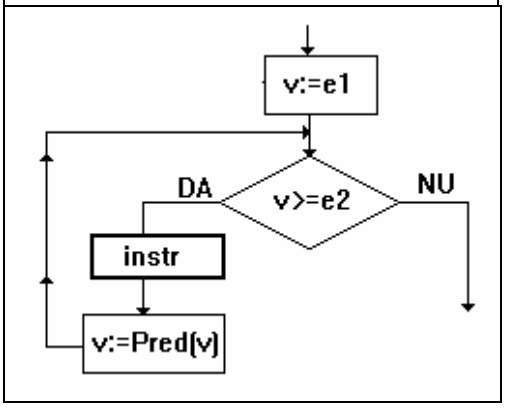
În ambele forme, `v` este o variabilă de tip scalar, numită `contor`, iar `e1` şi `e2` sunt două expresii de acelaşi tip cu `v`, numite respectiv *expresie inițială* şi *expresie finală*, iar `instr` este orice gen de instrucțiune, chiar şi o altă instrucțiune `for`.

Semantică

În primul caz, schema logică asociată instrucțiunii `for` - este:



Aşadar, `v` primeşte valoarea expresiei `e1`. Apoi, se verifică dacă s-a ajuns la valoarea `e2`. Dacă da, ciclul se încheie, iar dacă nu, atunci se execută instrucțiunea `instr`, apoi `v` creşte la succesorul său (în cazul numerelor întregi, de pildă, `v` creşte cu o unitate), apoi se reia verificarea condiției de continuare a ciclului.



În forma cu `downto`, avem un test invers de continuare, iar în loc de `Succ`, apare `Pred`.

Aşadar, se execută instrucțiunea `instr`, pentru `v` luând valori între `e1` şi `e2`, în primul caz crescător, în al doilea descrescător.

În primul caz, dacă `e2<e1`, atunci nu se execută nimic, trecându-se imediat după ansamblul instrucțiunii `for`, iar în al doilea caz, nu se execută nimic atunci când `e2>e1`.

Exemple

- Vom calcula suma `S=1+3+5+...+(2n-1)` cu ajutorul instrucțiunii `for`:

```
S:=0; for i:=1 to n do S:=S+2*i-1;
```

- Un dreptunghi format din caractere '#' între colțul `(x1,y1)` şi colțul `(x2,y2)` se poate realiza uşor aşa:

```
for x:=x1 to x2 do
     for y:=y1 to y2 do
       begin GoToXY(x,y); Write('#') end;
```

- Afişarea, în ordinea invers alfabetică, a tuturor literelor mari se poate face simplu:

`for c:='Z' downto 'A' do Write(c,',')`, unde avem `var c: Char`.

Erori frecvente

- Scrierea unui simbol "**;**" după `do` este o eroare făcută de unii, din neatenție. Ea nu produce eroare la compilare, dar produce mari neplăceri dacă nu e depistată la timp. De pildă, secvența de mai jos va afişa doar `5`:

```
for x:=1 to 5 do ;
     WriteLn(x)
```

- O altă greşeală, poate cea mai frecventă, este folosirea semnului "**=**" în loc de "**:=**", ca mai jos:

```
for i=1 to 10 do instr
```

- Există şi alte greşeli, cum ar fi folosirea tipurilor nescalare:

```
for d:=1 to Sqrt(n) do instr ;
for i:=n/2 downto 1 do instr
```

Program demonstrativ

Vom realiza următoarea "piramidă" a numerelor:

```
1
1 2
1 2 3
............
1 2 3 ... n
```

în care numărul `n` este citit de la tastatură.

Avem de afişat `n` linii, deci linia `i`, cu `i` de la `1` la `n`. Pe fiecare linie `i` avem de afişat numerele `1`,`2`,... şi `i`, deci numărul `j`, cu `j` de la `1` la `i`. La sfârşitul afişării unei linii, trebuie să trecem pe următoarea linie.

```
program Piramida;
var i,j,n: Integer;
begin
  Write('Dati n = '); ReadLn(n);
  for i:=1 to n do
      begin
        for j:=1 to i do
            Write(j:2); { adica j pe doua pozitii }
       WriteLn
      end;
  ReadLn
end.
```

Exerciții

1. Care este valoarea variabilelor `i` şi `p` după execuția următoarei secvențe de program:

```
a) p:=1; n:=1; for i:=2 to n do p:=p*i;
b) p:=1; n:=4; for i:=2 to n do p:=i*p;
c) i:=5; p:=0;
   while i<=3 do
        begin
          for j:=1 to i do p:=p+j;
           i:=i+1
        end;
```

2. Folosind instrucțiunea repetitivă `for` să se calculeze valoarea următorului produs:

$$P = (1-\frac{1}{2^2})(1-\frac{1}{3^2})...(1-\frac{1}{n^2}).$$

3. Indicați erorile sintactice din următoarele secvențe de instrucțiuni:

```
a) var a: Real; s: Real;
    begin
        s:=0;
        for a:=1 to 2 do s:=s+a;
        WriteLn(a)
    end.
b) var i,n: Integer; prim: Boolean;
     begin
         prim:=True;
         for i:=2 to Sqrt(n) do
              if n mod i = 0 then prim:=False;
        if prim then WriteLn('prim')
        else WriteLn('nu este prim')
    end.
c) var a,n,S: Integer; c: Char;
    begin
        a:=1; n:=26; S:=0;
        for c:=a to n do S:=S+c;
        WriteLn(c=,'c')
   end.
```

4. Să se calculeze suma `S=1+1·2+1·2·3+...+1·2·3·...·n`.

5. Să se calculeze suma `S=1-2+3-4+...± n`.

6. Să se calculeze suma `S=1-1·2+1·2·3-...±1·2·3·...·n`.

7. Să se afişeze "piramida" de numere de mai jos:

```
n n-1 n-2 ... 3 2 1
..................
3 2 1
2 1
1
```

8. Să se producă în difuzor un sunet crescător, apoi descrescător şi tot aşa până când se apasă o tastă.

9. Să se afişeze piramida de numere:

```
          1
        1 2 3
      1 2 3 4 5
    ..................
1 ..............(2n-1)
```

10. Să se simuleze deplasarea unei bile de la stânga la dreapta, pe linia din mijloc a ecranului. Aceeaşi problemă pentru sensurile dreapta-stânga, sus-jos, jos-sus.

4. Instrucțiunea repetitivă cu test final `REPEAT`

Necesitate

De multe ori este necesar ca un ciclu să aibă, la sfârşit, condiția de ieşire. De pildă, dacă vrem să deplasăm pe ecran o bilă (utilizând patru taste), până la acționarea tastei 's', vom scrie ceva de genul:

repetă
mişcă bila cu tastele corespunzătoare celor 4 sensuri
până când tasta apăsată este 's' .

Evident, chiar dacă, de la bun început, s-a apăsat tasta 's', oricum acțiunile din interiorul ciclului au loc măcar o dată, deoarece testul este la sfârşitul ciclului. De fapt, condiția de ieşire a unui astfel de ciclu este negația condiției de ieşire dintr-un ciclu `while-do`.

Sintaxă

Instrucțiunea repetitivă cu test final are următoarea formă:

```
repeat
  instr1;
  instr2;
  .........
  instrn
until cond
```

În cadrul ciclului putem avea mai multe instrucțiuni, de aceea nu e nevoie să folosim o instrucțiune compusă. Fireşte, `cond` este o expresie booleană, iar $instr_1$, ... $instr_n$ sunt instrucțiuni de orice tip, inclusiv instrucțiuni `repeat`.

Semantică

Sensul acestei instrucțiuni este: repetă instrucțiunile $instr_1$, ... , $instr_n$, până când este îndeplinită condiția `cond`. Instrucțiunile se execută *măcar o dată*. Folosind instrucțiunea `while`, putem exprima această instrucțiune astfel:

```
instr1 ;
.........
instrn ;
while not cond do
         begin
        instr1 ;  ........ instrn
         end;
```

Exemple

Să realizăm un program în care să se citească un număr întreg `n`, în funcție de care să se afişeze un anumit text. Vom restricționa citirea lui `n` în sensul introducerii doar a uneia din valorile 1 şi 2. Vom verifica, cu ajutorul unui ciclu `repeat-until`, dacă numărul introdus de la tastatură verifică sau nu condiția impusă.

```
program Anul;
var n: Integer;
begin
 repeat Write('Dati numărul (1/2) : '); ReadLn(n) until (n=1) or (n=2);
 if n=1 then WriteLn('Prima jumătate a anului.')
 else WriteLn('A doua jumătate a anului.'); ReadLn
end.
```

Erori frecvente

- Nu este vorba despre o eroare dacă scriem ceva de genul:

```
repeat
   begin
        instr1 ;
        ........
        instrn
   end
until cond;
```

însă prezența cuvintelor `begin` şi `end` nu este necesară.

- Unii începători confundă instrucțiunile *while* şi *repeat* între ele, scriind, de pildă:

```
repeat
    instr1 ;
    ........
    instrn
while cond  .
```

Program demonstrativ

Vom calcula cel mai mare divizor comun a două numere folosind **algoritmul lui Euclid**: "Pentru a obține *cmmdc* a două numere întregi `a` şi `b`, `b≠0`, împărțim `a` cu `b`; dacă restul împărțirii r_1 este zero, atunci `b` este `cmmdc`; dacă nu, împărțim pe `b` la restul împărțirii anterioare, r_1 , şi obținem restul r_2; apoi împărțim pe r_1 la r_2 şi obținem un nou rest r_3 ş.a.m.d.. Ultimul rest nenul este *cmmdc* al celor două numere.".

Justificarea algoritmului lui Euclid este dată de proprietatea: `cmmdc(a,b)=cmmdc(b,a mod b)`.

Programul următor rulează ciclic, pentru mai multe perechi de valori ale lui `a` şi `b`, până când răspunsul la întrebarea de continuare este '`n`'. Fireşte, programul dă eroare la execuție dacă `b=0` la început, iar acest lucru poate fi evitat dacă se foloseşte un ciclu `while-do` în locul ciclului `repeat-until` interior. Vă propunem dumneavoastră să realizați acest lucru.

```
program Euclid_CelMaiMareDivizorComun;
var a, b, deimp, imp, rest, cmmdc: Integer; raspuns: Char;
begin
     repeat
       WriteLn('Dati cele doua numere !'); ReadLn(a,b);
       deimp:=a; imp:=b;
       repeat
        rest:=deimp mod imp; deimp:=imp; imp:=rest
       until imp=0;
       cmmdc:=deimp; WriteLn('C.m.m.d.c. = ',cmmdc); WriteLn;
       WriteLn('Repetam ? șd/nț'); ReadLn(raspuns)
     until raspuns='n'
end.
```

Exerciții

1. Să se calculeze, folosind instrucțiunea `repeat`, suma `S=2+4+6+...+(2n)`, unde `n≥1`.

2. Să se producă în difuzor câte un sunet aleator, până se acționează o tastă.

3. Rescrieți programul de la problema 4 din lecția 2, pentru a folosi instrucțiuni `repeat` în locul instrucțiunilor `while`, acolo unde utilizarea instrucțiunilor `while` este incomodă.

5. Instrucțiunea de selecție multiplă CASE

Necesitate

O familie de tineri căsătoriți doreşte să-i cumpere noului născut ceva de îmbrăcat, pentru atunci când copilul va fi ceva mai măricel. El zice: "dacă e fată îi luăm o fustiță, iar dacă e băiat o pereche de pantalonaşi." Această problemă se poate soluționa cu ajutorul instrucțiunii de decizie `if-then-else`. Dar ea întreabă: "dar dacă sunt două fetițe gemene sau doi băieți gemeni sau chiar o fetiță şi un băiețel gemeni ?". Ei bine, problema se complică, aşa încât trebuie prevăzute mai mult de două cazuri. Soții s-ar putea folosi de o descriere de genul:

```
în caz că noul născut este
   o fată: cumpărăm o fustită;
   un băiat: cumpărăm o pereche de pantalonasi;
```

```
    două fete: cumpărăm două fustite;
    doi băieti: cumpărăm două perechi de pantalonasi;
    o fată si un băiat: cumpărăm o fustită si o pereche de pantalonasi
gata;
```

Dar e posibil să iasă trei gemeni sau patru, două fete şi doi băieți, aşa încât cei doi vor cumpăra scutece, în orice alt caz:

```
în caz că noul născut este
    o fată: cumpărăm o fustită;
    un băiat: cumpărăm o pereche de pantalonasi;
    două fete: cumpărăm două fustite;
    doi băieti: cumpărăm două perechi de pantalonasi;
    o fată si un băiat: cumpărăm o fustită si o pereche de pantalonasi
altfel cumpărăm scutece
gata;
```

Această problemă şi altele asemănătoare sunt soluționate de instrucțiunea de selecție multiplă: `case`.

Sintaxă
Această instrucțiune poate lua mai multe forme, care pot fi prezentate unitar ca mai jos, în care ramura cu `else` este opțională. Instrucțiunea constă dintr-un *selector* (o expresie de tip ordinal) şi o *listă de instrucțiuni*, fiecare din ele prefixată de un *caz*:

```
case expresie of
   caz1: instructiune1;
   ...................
   caz2: instructiunen;
end
```

sau:

```
case expresie of
   caz1: instructiune1;
   ...................
   caz2: instructiunen
 else instructiunen+1
end
```

Un "caz" constă dintr-unul sau mai multe constante sau domenii, separate prin virgule, şi având acelaşi tip cu selectorul. Ramura cu `else` este opțională.
De exemplu, pentru numere întregi, o constantă ar fi `23`, iar un domeniu este o constantă urmată de "`..`" şi altă constantă: `20..30`, de exemplu.

Semantică
Se evaluează expresia selector şi, în funcție de valoarea sa, se execută instrucțiunea sau setul de instrucțiuni corespunzătoare; în cazul în care nu-i corespunde niciuna se execută, dacă există, instrucțiunea de pe ramura *else*.

Exemple
- În exemplul următor se precizează felul unui anumit caracter `Ch`:

```
case Ch of
   'A'..'Z', 'a'..'z': WriteLn('Literă');
   '0'..'9':  WriteLn('Cifră');
   '+', '-', '*', '/': WriteLn('Operator')
 else WriteLn('Caracter special')
 end;
```

- În continuare, să realizăm un program care să afişeze numele anotimpului corespunzător unui număr citit de la tastatură. Prima variantă este realizată folosind instrucțiunea `if`, iar cea de a doua, mult mai elegantă, utilizează instrucțiunea `case`.

```
program Anotimpuri1;
var nr: Integer; nume: String;
begin
  Write('Dati numarul anotimpului = '); ReadLn(nr);
   if nr = 1 then WriteLn('Iarna')
   else if nr = 2 then WriteLn('Primavara')
        else if nr = 3 then WriteLn('Vara')
             else if nr = 4 then WriteLn('Toamna')
                  else WriteLn('Nu e un numar corect !'); ReadLn
end.
```

```
program Anotimpuri2;
var nr: Integer; nume: String;
begin
  Write('Dati numarul anotimpului = '); ReadLn(nr);
   case nr of  1: WriteLn('Iarna');
               2: WriteLn('Primavara');
               3: WriteLn('Vara');
               4: WriteLn('Toamna')
              else WriteLn('Nu e un numar corect !')
   end; ReadLn
end.
```

Erori frecvente

- De remarcat că un "punct şi virgulă" înainte de `else` sau înainte de `end` nu este considerat eroare.
- De multe ori se greşeşte încercând să se folosească instrucțiunea `case` cu selector de tip `String`, deci neordinal:

```
case nume of
        'Iarna': WriteLn('Primul anotimp');
        'Primavara': WriteLn('Al doilea anotimp');
end;
```

Ar fi de dorit să dispunem de o asemenea instrucțiune de selecție multiplă, dar din păcate `case` funcționează doar pentru tipurile scalare, deci va trebui să recurgem, în asemenea situații, la mai multe instrucțiuni `if`.

Program demonstrativ

Vom scrie un program care să pună întrebarea: "*În ce an s-a născut Eminescu ?*" şi. dacă răspunsul este corect (*1850*), atunci să afişeze "*Foarte bine*", altfel un text corespunzător.

```
program TestEminescu;
var an: Integer;
begin
  Write('~n ce an s-a născut Eminescu ? '); ReadLn(an);
   case an of
     1850: WriteLn('Foarte bine !');
     1849,1851: WriteLn('Aproape bine !');
     1845..1848: WriteLn('Cu ',1850-an, ' ani mai târziu !');
     1852..1855: WriteLn('Cu ',an-1850, ' ani mai devreme !');
     1800.1844, 1856..1899: WriteLn('Aţi nimerit secolul !')
    else
      WriteLn('Cultura dumneavoastră are lacune grave !')
   end;
   ReadLn
end.
```

6. Exerciții recapitulative

1. Se citeşte un şir de n numere reale. Să se afişeze valoarea minimă şi cea maximă din şir, determinate simultan.
2. Se citeşte un şir de numere întregi pâna la întâlnirea numărului 0. Să se calculeze media aritmetică a numerelor din şir.
3. Se citesc 3 numere naturale n, p şi k, apoi un şir de n numere naturale. Câte dintre acestea, împărțite la p dau restul k ?
4. Să se calculeze produsul a două numere naturale prin adunări repetate.
5. Efectuați împărțirea întreagă a două numere, fără a utiliza operatorii mod şi div.
6. Se citeşte un număr natural. Câte cifre conține ?
7. Un număr se numeşte "palindrom" dacă citit invers este acelaşi număr. Să se verifice dacă un număr este sau nu palindrom.
8. Să se afişeze toate numerele prime mai mici sau egale cu un numar m dat.
9. Să se afişeze primele n numere prime care au suma cifrelor ≤ m.
10. Să se afişeze toate numerele de 3 cifre care, citite invers, sunt tot numere prime.
11. Calculați şi afişați suma: 1/(1×2) + 1/(2×3) + 1/(3×4) + ... + 1/(*n*×(*n*+1)).
12. Să se verifice dacă un număr natural este sau nu pătrat perfect.
13. Să se scrie un program care să rezolve câte o ecuatie de gradul II, până utilizatorul programului nu mai vrea acest lucru.
14. Să se verifice dacă un număr natural este sau nu cub perfect.
15. Să se afişeze toate numerele de forma $a^2 + b^3$, cu $1 \le a \le 5$ şi $1 \le b \le 5$.
16. Să se determine toate triunghiurile diferite cu laturi numere întregi pozitive şi perimetru p.
17. Să se listeze toate numerele ≤ n, a căror sumă a cifrelor este divizibilă prin 5.
18. Să se transforme un număr din baza 10 în baza p < 10.
19. Să se transforme un număr din baza p < 10 în baza 10.
20. Să se scrie un program care afişează:

```
1
1 2
1 2 3
........
1 2 3 .. n
1 2 3 .. n
........
1 2 3
1 2
1
```

21. Să se transforme un număr din baza p în baza q, unde p,q ≤ 10.
22. Se citesc numere naturale până la întâlnirea numărului 0. Să se afişeze soluțiile tuturor ecuațiilor de gradul I ax+b=0, unde a şi b sunt toate perechile de numere citite consecutiv, în care b este divizibil prin a.
23. Se citesc numere naturale până la introducerea unui număr real. Să se calculeze suma S a tuturor numerelor citite, precum şi câtul şi restul împărțirii lui S la suma cifrelor lui S.
24. Se citesc numere naturale până la întâlnirea numărului 12. Să se afişeze toate tripletele de numere citite consecutiv, în care al treilea număr este restul împărțirii primului la al doilea.
25. Se citesc numere naturale până la întâlnirea numărului 0. Să se afişeze toate tripletele de numere citite consecutiv, în care al treilea număr este media aritmetică (geometrică) dintre primul şi al doilea.
26. Se citesc numere naturale până la întâlnirea numărului 0. Să se afişeze toate perechile de numere citite consecutiv, cu proprietatea că al doilea număr este egal cu suma cifrelor primului număr.

27. Se citesc numere naturale până la întâlnirea numărului `0`. Să se afişeze toate perechile de numere citite consecutiv, cu proprietatea că al doilea număr reprezintă restul împărțirii primului număr la suma cifrelor sale.
28. Se citesc numere naturale până la întâlnirea numărului `0`. Să se afişeze toate perechile de numere citite consecutiv, cu proprietatea că al doilea număr reprezintă numărul de apariții ale cifrei `3` în pătratul primului.
29. Se citesc numere naturale până la întâlnirea numărului `0`. Să se afişeze toate perechile de numere citite consecutiv, cu proprietatea că al doilea număr reprezintă pătratul numărului de apariții ale cifrei *1* în primul.
30. Se citesc numere naturale până la întâlnirea numărului `0`. Să se afişeze toate tripletele de numere citite consecutiv, cu proprietatea că al treilea număr este suma dintre primul şi al doilea.
31. Se citesc numere reale până la întâlnirea numărului `0`. Să se afişeze toate tripletele de numere citite consecutiv, cu proprietatea că ele pot reprezenta laturile unui triunghi.
32. Se citesc numere reale până la întâlnirea numărului `0`. Să se afişeze toate tripletele de numere citite consecutiv, cu proprietatea că ele pot reprezenta laturile unui triunghi isoscel.
33. Se citesc numere reale până la întâlnirea numărului `0`. Să se afişeze toate tripletele de numere citite consecutiv, cu proprietatea că ele pot reprezenta laturile unui triunghi dreptunghic.
34. Se citesc numere reale până la întâlnirea numărului `0`. Să se afişeze soluțiile tututor ecuațiilor de gradul 2 $ax^2+bx+c=0$, unde `a`, `b` şi `c` sunt toate tripletele de numere citite consecutiv, în care $b^2-4ac=0$.
35. Se citesc numere reale până la întâlnirea numărului `0`. Să se afişeze soluțiile tututor ecuațiilor de gradul 2 $ax^2+bx+c=0$, unde `a`, `b` şi `c` sunt toate tripletele de numere citite consecutiv, în care $b^2-4ac>0$.
36. Se citesc numere naturale până la întâlnirea numărului `0`. Să se afişeze toate perechile de numere (`n1`,`n2`) citite consecutiv, cu proprietatea că `n1>n2` şi suma cifrelor lui `n1` este mai mică decât suma cifrelor lui `n2`.
37. Să se găsească toate reprezentările posibile ale numărului natural `n` ca sumă de numere naturale consecutive.
38. Următorul program vrea să determine dacă un numar este prim sau nu. Este acest program corect ? De ce ? Verificati programul pentru numerele `26`,`27`,`32` !

```
program PrimGresit;
var  d,x: LongInt; prim: Boolean;
begin
     Write('x='); ReadLn(x);
     for d:=2 to Trunc(Sqrt(x)) do
          if x mod d = 0 then prim:=False
          else prim:=True;
     if prim then WriteLn(x,' este prim')
     else WriteLn(x,' nu este prim');
     ReadLn
end.
```

39. Care din următoarele programe este mai rapid ? De ce ? Verificați pentru numărul `2123456789` ! Comparați programele cu programul **TestNumarPrim**.

```
program Prim1;
var  d,x: LongInt; prim: Boolean;
begin
     Write('x='); ReadLn(x);
     prim:=True; d:=2;
     while (d<=Trunc(Sqrt(x))) and prim do
           if x mod d = 0 then prim:=False
```

```
            else if d=2 then d:=d+1 else d:=d+2;
      if prim then WriteLn(x,' este prim')
      else WriteLn(x,' nu este prim');
      ReadLn
end.
program Prim2;
var  d,x: LongInt; prim: Boolean;
begin
      Write('x='); ReadLn(x);
      prim:=True;
      for d:=2 to Trunc(Sqrt(x)) do
           if x mod d = 0 then prim:=False;
      if prim then
          WriteLn(x,' este prim')
      else
         WriteLn(x,' nu este prim');
      ReadLn
end.
```

40. Următoarele două programe rezolvă aceeaşi problemă: determină toate numerele `i`, `j`, `k` care pot reprezenta laturile unui triunghi de perimetru `p`, `p` dat de la tastatură. Una din metode este mai rapidă. Care din ele ? De ce ? Verificați ambele programe pentru `p = 100`, apoi pentru `p = 1000` !

```
program Laturi_A;
var i,j,k,p: Integer;
begin
  Write('Dati perimetrul: p = '); ReadLn(p);
   for i:=1 to p do
   for j:=1 to p-i do
       begin
         k:=p-(i+j);
         if (i<j+k) and (j<i+k) and (k<i+j) then
             WriteLn(i, ',', j, ',', k)
       end;
  ReadLn
end.
```

```
program Laturi_B;
var i,j,k,p: Integer;
begin
  Write('Dati perimetrul: p = '); ReadLn(p);
   for i:=1 to p do
       for j:=1 to p do
           for k:=1 to p do
              if (i+j+k=p) and (i<j+k) and (j<i+k) and (k<i+j) then
                 WriteLn(i, ',', j, ',', k);
  ReadLn
end.
```

41. Ridicarea la putere. Următoarele două programe rezolvă aceeaşi problemă: calculează n^k, unde `n` şi `k` sunt două numere întregi pozitive. Al doilea program este mai rapid, el necesitând mai puține înmulțiri. Verificați programele pentru `n=1` şi `k=90000`.

```
program n_la_puterea_k__A;
var i,n,k,p: LongInt;
begin
     Write('Dati n = '); ReadLn(n); Write('Dati k = '); ReadLn(k);
     p:=1; for i:=1 to k do p:=p*n;
     WriteLn('Rezultat : ',p); ReadLn
end.
```

```
program n_la_puterea_k__B;
var i,n,k,p: LongInt;
```

```
begin
     Write('Dati n = '); ReadLn(n);
     Write('Dati k = '); ReadLn(k);
     p:=1;
     while k>0 do
           if k mod 2 = 0 then begin n:=n*n; k:=k div 2 end
           else begin p:=p*n; k:=k-1 end;
     WriteLn('Rezultat : ',p); ReadLn
end.
```

42. **Numerotarea paginilor unei cărți**. Pentru numerotarea paginilor unei cărți sunt necesare `n` cifre. Câte pagini conține cartea ?

Lecția 4
TIPURI DE DATE DEFINITE DE PROGRAMATOR

1. Tipul enumerare şi tipul subdomeniu

Până acum am folosit doar tipurile de date pe care le numeam predefinite. Astfel, o variabilă putea fi doar un număr (întreg sau real), un caracter, un şir de caractere sau o valoare logică. Dar să presupunem că vrem să lucrăm la un nivel mai abstract, de exemplu cu semafoare electrice. Astfel, am avea nevoie de un nou tip de date, cu numele `semafor`, iar o variabilă `s`, de acest tip, s-ar declara în mod obişnuit, prin `var s: semafor`.

Astfel, vom intra împreună în minunata lume a tipurilor de date definite de programator. Un astfel de tip se declară prin:

```
type identif_de_tip = definitie_de_tip;
```

În partea stângă avem un identificator, neîntâlnit încă în program, iar în partea dreaptă avem o definiție de tip, aceasta diferind de la un tip la altul.

Tipul enumerare

Un tip enumerare se defineşte printr-o listă de identificatori *abstracți*, separa]i prin virgule şi încadrați de paranteze rotunde.

```
type id_tip_enumerare=(id1, id2,..., idn)
```

Exemple:

-
```
type semafor = (rosu, galben, verde);
```
sau:
```
type saptamana = (luni, marti, miercuri, joi,
                  vineri, sambata, duminica);
```

Putem scrie pe scurt, ca şi la `const` sau `var` :

```
type semafor = (rosu, galben, verde);
     saptamana = (luni, marti, miercuri, joi,
                  vineri, sambata, duminica);
```

Fireşte, dacă avem declarațiile de variabile: `var s: semafor; zi: saptamana;` trebuie să înțelegem că `s` poate avea una din valorile celor trei identificatori (ce numesc cele trei culori ale semaforului), iar `zi` poate fi una din zilele saptămânii.

Elementele unei enumerări pot fi, evident, numărate, deci vom avea funcțiile `Ord`, `Pred` şi `Succ`.

Fie declarațiile următoare:
```
type saptamana = (luni, marti, miercuri, joi,
                  vineri, sambata, duminica);
var zi: saptamana; n: Integer;
```

În urma atribuirilor de mai jos:
```
zi:=miercuri; n:=Ord(Succ(zi));
```

`zi` va avea valoarea `miercuri`,

iar `n` va fi `Ord(Succ(miercuri))=Ord(joi)=3` (numerotarea începe de la `0`).

Erori frecvente:

- Se greşeşte adesea încercând să se afişeze variabile sau constante de tip enumerare, uitându-se că ele sunt valori abstracte şi nu şiruri de caractere sau identificatori pentru valori de alt tip. `WriteLn(miercuri)` va da eroare, iar `WriteLn('miercuri')` va afişa şirul `miercuri`, însă acest şir nu are nici o legătură cu identificatorul `miercuri`.
- De multe ori se greşeşte când se declară tipul, punându-se simbolurile pentru atribuire în loc de semnul de egalitate, după `type`:
```
type culoare := (rosu, galben, albastru, verde, violet);
```

Utilizare:

Datele de tip enumerare pot fi folosite în programe pentru a face mai sugestive anumite lucruri. De pildă, dacă facem un joc de poker, iar jucătorul va fi, pe rând omul şi calculatorul, atunci alternarea lor se descrie astfel:

```
type jucator = (om, calculator);
var juc: jucator;
.....
if  juc = om then  juc := calculator
else juc := om;
```

Observație: Nu e nevoie să definim tipul `jucator`, putem scrie direct:

```
var juc: (om, calculator) !
```

Tipul subdomeniu

De multe ori dorim să folosim doar numere naturale în program, deci nu tot domeniul `Integer`, ci doar un subdomeniu al său. Acest lucru e posibil, dacă declarăm: `type Natural = 0..200;`. Aceasta reprezintă faptul că s-a definit un nou tip de date, numit `Natural`, care reprezintă toate numerele întregi pozitive mai mici sau egale cu 200. Dacă vrem să extindem la maxim în dreapta această arie, vom scrie: `type Natural=0..MaxInt;`. Acum putem lucra cu numere întregi considerate doar naturale: `var a,b,cmmdc: Natural;`. Dacă vrem să lucrăm doar cu litere mari, adică cu acest subset al tipului `Char` vom defini: `type Litere = 'A'..'Z';`. Dacă vrem să lucrăm cu un şir de maxim 20 de elemente, memorate într-un vector (vezi lecția 1), vom defini un subdomeniu de genul: `type indice=1..20;`. Prin declarația: `var i: indice;` vom marca faptul că, deşi `i` este un număr întreg, el nu poate lua chiar orice valoare, ci doar între `1` şi `20`.

Deci tipul subdomeniu se define[te astfel:

```
type id_tip_subd = const_1..const_2
```

De fapt, chiar şi alte trei tipuri întregi, predefinite în Turbo-Pascal, sunt definite ca subdomenii:

```
type ShortInt = -128..127;
     Byte = 0..255;
     Word = 0..65535;
```

Astfel, dacă avem `var x: Byte;` o atribuire de genul `x:= - 10` este eronată !

Observație: Nu e nevoie să ataşăm neapărat un nume unui tip subdomeniu, putem scrie, de pildă, `var i: 1..20`. Spunem că tipul este *anonim*.

Fireşte, cum tipul `Real` nu este scalar, nu vom putea defini ceva de genul: `type RealPozitivMic = 0..123.5`.

Exerciții

1. Definiți tipuri enumerare pentru lunile anului, pentru anotimpuri, pentru disciplinele de învățământ.

2. Definiți tipuri subdomeniu pentru literele mici ale alfabetului, pentru cifre.

3. Definiți o enumerare pentru zilele săptămânii şi un subdomeniu al acestora pentru zilele lucrătoare.

4. Definiți tipuri de date pentru litere mari, litere mici, cifre, cifre pare, cifre impare.

5. Definiți un tip de date enumerare pentru culorile cele mai uzuale şi unul ca subdomeniu al acestuia pentru culorile deschise.

2. Tablouri unidimensionale (vectori)

În primul capitol am vorbit despre nevoia, în multe probleme, de a lucra cu şiruri (de o lungime necunoscută) de variabile de acelaşi tip. De pildă, pentru a memora înălțimile unor elevi, exprimate în centimetri, vom avea nevoie de un şir de numere întregi, iar pentru a memora numele acestor elevi, de un şir de date de tip `String`.

Un şir de elemente de acelaşi tip se mai numeşte şi **vector** sau **tablou unidimensional**.
Dacă însă dorim să facem un program de şah, va trebui să punem tablei te joc în corespondență o **matrice** (un **tablou bidimensional**), adică o suprafa]ă dreptunghiulară organizată pe linii şi pe coloane.
Este clar că un tablou grupează date de acelaşi tip, a căror referire se va face prin poziția lor în şir, respectiv prin linia şi coloana din matrice.

În mod similar, putem să ne imaginăm şi tablouri cu mai multe dimensiuni. De pildă, pentru a memora coordonatele - presupuse numere întregi - ale unor puncte din spațiul tridimensional vom avea nevoie de un tablou cu trei dimensiuni.
Astfel de tipuri de date, ca şi cele ce vor fi prezentate în continuarea lecției se numesc **tipuri structurate**, deoarece, după cum se vede, o dată de acest tip are o structură mai complexă.

Sintaxă
Putem să definim un vector de elemente de acelaşi tip, în felul următor:

```
type id_de_tip_tablou = array[id_de_tip_scalar] of  id_de_tip
```

Exemple. Erori frecvente

- `type vector = array[1..20] of Integer` ar însemna un şir de elemente numere întregi, numerotate de la `1` la `20`. Fie acum o variabilă de acest tip: `var x:vector;`. Al treilea element din `x` se referă prin `x[3]`, deci a `i`-a componentă se va specifica prin `x[i]`.
Dar putem să definim şi tablouri cu elemente numerotate, de pildă, de la `a=-5` la `b=5` (tabloul va avea `b-a+1` elemente): `type vec = array[-5..5] of Integer;`
- `type tablou = array['a'..'z'] of Real` ar însemna un şir de numere reale, "numerotate" de la 'a' la 'z', deci marginile sunt caractere. Dacă vectorul nostru `var x: tablou` ar fi : (`12.5,100,4.45,-8.25,0,9.3,-56.456,`...), atunci am putea spune că a 'd'-a componentă este `x['d']=-8.25` !
- Dar am putea defini chiar: `type vector_lung=array[Char] of Integer` ceea ce ar însemna un şir de elemente numere întregi, numerotate de la caracterul cu codul `ASCII 0` până la caracterul cu codul `ASCII 255`, care este ultimul !
- Fireşte, putem folosi chiar şi tipul `Boolean` pentru a defini tablouri, chiar dacă sună ciudat: `type taboul_ciudat=array[Boolean] of String`. Am putea avea un tablou: `var nume: tablou_ciudat` şi am putea vorbi de a `False`-a componentă sau a `True`-a componentă a sa, adică de prima sau de a doua !
- Ultimele exemple prezentate, deşi nu sunt tocmai interesante şi utile, sunt corecte. Am putea merge chiar până la defini]ii de genul `type ciudat_de_tot=array[semafor] of semafor`. În continuare, vom lucra cu tablouri de numere întregi sau reale, sau având elemente de tip `Boolean`, `Char` sau `String` , tablouri care modelează exemplele necesare din realitate.
- Dacă vom avea definit `type vector=array[1..20] of Integer`, vom nota o variabilă de acest tip în general cu `x`. Componenta sa a `i`-a va fi, evident `x[i]`, iar noi vom utiliza efectiv, doar primele `n` componente (deci $1 \le n \le 20$).
- Putem scrie ceva de genul `type vector=array....` şi `var x: vector` pe scurt prin `var x: array.....` .
- De multe ori se greşeşte în definiția tipurilor tablou, punându-se o variabilă `n` acolo unde ar trebui să apară o valoare constantă: `type vector = array[1..n] of...` .
- Unii pun mai mult de două puncte între cele două constante care definesc domeniul indicelui: `type vector = array[1....10] of Integer`, ceea ce evident este greşit.
- A `i`-a componentă este `x[i]`, după cum am spus, nicidecum `x(i)` sau `xi` sau, cum scriu unii pe foaia de hârtie, x_i , aşa cum scriam şi noi în prima lecție.

- Unii începători greşesc încurcând tipul cu variabila (pe care uită să o mai declare !), de exemplu ei scriu: `type vector = array[1..20] of Integer`; şi apoi vor să se refere la a `i`-a componentă dintr-un astfel de vector prin `vector[i]` !

Operații simple cu vectori

O variabilă de tip tablou nu poate fi citită sau scrisă în întregime, dar - dacă în primele versiuni de Pascal, nu se putea face nici atribuiri ‘ntre variabile de *acelaşi tip* tablou, în Turbo-Pascal 7.0 acest lucru e realizabil.

În orice caz, e mai bine să lucrăm pe componente. Cu componentele unui tablou se pot face toate operațiile ce se pot face cu orice variabilă de acel tip (afişare, citire, atribuire etc.).

În cele ce urmează vom prezenta secvențe de program care vor face diferite prelucrări asupra unui vector `x` de numere reale. Să considerăm, aşadar, că avem declarațiile:

```
type domeniu=1..20;  vector: array[domeniu] of Real;
var x, y: vector; n,i: domeniu;
```

Vom prezenta modul de citire sau afişare a unui astfel de vector. Pentru a realiza aceste operații cu vectori de alt tip, modificările nu sunt mari.

1. Citirea unui vector

Aceasta înseamnă citirea numărului `n` de componente utilizate efectiv, precum şi a tuturor celor `n` componete. Citirea numărului de componente este simplă: `Write('Dati n = '); ReadLn(n);`. Apoi trebuie să citim toate componentele `x[1], x[2], ..., x[n]`, restul componentelor rămânând nefolosite. De fapt, avem de citit componenta numărul `i`, cu `i` de la 1 la `n`. Deci, putem scrie:

```
for i:=1 to n do
    begin
      Write('Dati x[',i,']=');
      ReadLn(x[i])
    end;
```

2. Scrierea unui vector

Când trebuie să afişăm vectorul, adică toate componentele sale cu care se lucrează efectiv, numărul acestora este cunoscut. Aşadar, vom afişa, în general, componenta a `i`-a, cu `i` de la `1` la `n`. Afişarea poate fi făcută în două feluri:

- fiecare element pe un rând (folosită mai ales atunci când avem vectori de şiruri de caractere):

```
for i:=1 to n do
     WriteLn(x[i]);
```

- toate elementele pe acelaşi rând, despărțite de virgule şi/sau spații (în cazul valorilor numerice):

```
for i:=1 to n do
    Write(x[i],', ');
WriteLn;
```

Astfel, un program care prelucrează vectori va avea, în general, trei părți. Prima parte constă în citirea vectorului, cea de a doua este prelucrarea propriu-zisă a vectorului. Partea finală, dacă e necesară, va fi afişarea rezultatului obținut: vectorul inversat, suma componentelor sale ş.a...

Exemplu: Să citim un vector şi să-l afişăm în ordine inversă, deci, mai întâi componenta `x[n]`, apoi `x[n-1]` ş.a.m.d., ultima valoarea afişată fiind `x[1]`. Aceasta nu înseamnă că vectorul este şi inversat, el este doar *afişat invers* !

```
program CuVector;
type domeniu=1..20; vector = array[domeniu] of Real;
var x: vector; i,n: domeniu;
begin
  { citirea vectorului }
```

```
    Write('Dati n = '); ReadLn(n);
    for i:=1 to n do
        begin Write('Dati x[',i,']='); ReadLn(x[i]) end;
    { afisare inversa }
    for i:=n downto 1 do Write(x[i]:5:2,', ');
    { x[i] este numar real ! }
    WriteLn; ReadLn
  end.
```

3. Inversarea unui vector într-un alt vector

De această dată vom inversa vectorul `x` în vectorul `y`, de acelaşi tip cu `x`, adică vom avea: în componenta `y[1]` pe `x[n]`, în componenta `y[2]` pe `x[n-1]` , ..., în componenta `y[n]` pe `x[1]`. Observăm că întotdeauna suma celor doi indici de poziție (din `x` şi `y`) este aceeaşi şi anume `n+1`; prin urmare, în cazul general, în componenta `y[i]` vom pune valoarea lui `x[n+1-i]`.

```
for i:=1 to n do y[i]:=x[n+1-i]
```

4. Inversarea unui vector în el însuşi

Nu la fel de simplu stau lucrurile atunci când vrem să inversăm vectorul `x`, dar rezultatul să fie tot în `x`. Astfel, ceva de genul: `for i:=1 to n do x[i]:=x[n+1-i]` nu este corect, deoarece după ce se trece de jumătatea din cele `n` componente, celelalte se copiază din prima jumătate !

Nici în felul următor - care interschimbă pe `x[i]` cu `x[n+1-i]` - nu e corect:

```
for i:=1 to n do
   begin
     aux:=x[i];  x[i]:=x[n+1-i];
     x[n+1-i]:=aux
   end;
```

deoarece aceasta va avea ca efect următoarele: va inversa prima componentă cu ultima, pe a doua cu penultima ş.a.m.d., dar după ce se trece de mijloc, se reinversează, aşa încât vectorul se readuce la forma inițială. Prin urmare, va trebui să ne oprim la *mijlocul* său. Prin *mijloc* vom înțelege `n div 2`, indiferent dacă `n` este par sau impar. Aşadar, va trebui să parcurgem vectorul până la `n div 2`, interschimbând pe `x[i]` cu `x[n+1-i]`:

```
for i:=1 to n div 2 do
     begin aux:=x[i];  x[i]:=x[n+1-i];  x[n+1-i]:=aux end;
```

Fireşte, `aux` va fi o variabilă de acelaşi tip cu `x[i]`, deci dacă `x` este un vector de numere reale, atunci şi `aux` va fi o variabilă reală..

5. Determinarea minimului unui vector

Această problemă, prezentată şi în primul capitol, se rezolvă astfel: considerăm minim primul element; apoi parcurgem restul vectorului şi, ori de câte ori găsim un element mai mic, actualizăm minimul la valoarea acelui element.

```
minim := x[1];
for i:=2 to n do if x[i]<minim then minim := x[i];
```

În acest moment, la sfârşitul ciclului, variabila `minim` (de acelaşi tip ca şi componentele lui `x`) conține cel mai mic element din vector.

6. Determinarea simultană a minimului şi a maximului dintr-un vector

Pentru a realiza aceste lucruri simultan, se procedează după cum urmează:

```
minim := x[1]; maxim := x[1];
for i:=2 to n do
     begin
        if  x[i]<minim then minim := x[i];
        if x[i]>maxim then maxim := x[i]
     end;
```

Evident, dacă vectorul conține doar un singur element, acela va fi şi maxim şi minim.

7. Însumarea componentelor unui vector
Suma componentelor se calculează ca orice sumă. Se pleacă cu suma nulă. Apoi, la fiecare pas `i` (`i` de la `1` la `n`), suma creşte cu elementul curent, care este `x[i]`.

```
Suma := 0; for i:=1 to n do Suma := Suma + x[i];
```

Propunem cititorului să calculeze suma pătratelor componentelor unui vector.

8. Afişarea elementelor pare dintr-un vector de numere întregi
Să considerăm `var x: array[1..20] of Integer` şi `var n,i: Integer`. În rezolvarea acestei probleme, trebuie să nu uităm că este vorba de o *afişare*. Deci se parcurge vectorul, dar nu se afişează toate numerele, ci doar acelea care sunt pare:

```
for i:=1 to n do if not odd(x[i]) then WriteLn(x[i])
```

9. Afişarea elementelor impare de pe poziții pare
Vectorul are componente întregi. Considerăm aceleaşi variabile ca şi la problema anterioară. Dacă spunem *poziție*, trebuie să ne gândim la indicele `i`, iar dacă spunem *element*, trebuie să ne gândim la elementul de pe poziția `i`, deci la `x[i]`. Aşadar, va trebui să avem `not odd(i)` şi `odd(x[i])`:

```
for i:=1 to n do
   if (not odd(i)) and odd(i) then WriteLn(x[i])
```

10. Numărarea elementelor negative şi a celor pozitive dintr-un vector
Să notăm prin `NN` şi `NP` cele două numere. Acestea se comportă ca două contoare, inițial nule. De fiecare dată când găsim în şirul de numere un element negativ se incrementează `NN`: `NN:=NN+1`, altfel `NP`.

Turbo-Pascal ne pune la dispoziție o procedură de incrementare mai rapidă: **`Inc`**`(NN)`. De fapt, putem incrementa o variabilă întreagă `x` cu orice valoare întreagă `a` prin `Inc(x,a)` (ceea ce înseamnă `x:=x+a`). De asemenea, `Inc(x)` este echivalent cu `Inc(x,1)`. Există o procedură similară pentru decrementarea valorilor întregi: **`Dec`**`(x,a)`⇔ `x:=x-a`, iar `Dec(x)`⇔`Dec(x,1)`.

Fireşte am putea scrie şi, mai general, `x:=Succ(x)`, respectiv `x:=Pred(x)` în loc de `x:=x+1`, respectiv `x:=x-1`.

Revenind la problema inițială, iată soluția ei:

```
NN:=0; NP:=0;
for i:=1 to n do
     if x[i]<0 then Inc(NN)
     else Inc(NP);
WriteLn('Exista ',NN, ' elemente negative.');
WriteLn('Exista ',NP, ' elemente pozitive. ')
```

11. Căutarea secvențială a unui element într-un vector
Prezentată şi în primul capitol, această problemă necesită, pentru rezolvarea ei, o parcurgere a vectorului, de la prima poziție, până la sfârşit sau până s-a găsit elementul căutat. *Găsirea* elementului căutat este marcată de o variabilă logică `gasit`, poziționată inițial pe fals. Fie `x` vectorul, iar `a` elementul căutat.

```
gasit:=False;
i:=1;
while (i<=n) and (not gasit) do
         if x[i]=y then gasit:=True
         else Inc(i)
```

12. Inserarea unui element într-un vector
Să inserăm, pe poziția `p` într-un vector, un element nou `m`. Pentru aceasta, vom deplasa elementele de pe pozițiile de la `p` la `n`, unde `n` este numărul de elemente ale vectorului, cu o poziție mai încolo. Deplasarea se va face de la poziția ultimă înspre poziția `p`. În

final vom pune elementul m pe poziția p. Fireşte, numărul de elemente ale vectorului va creşte cu o unitate: n:=n+1.

```
for i:=n+1 downto p+1 do
     x[i]:=x[i-1];
x[p]:=m;
n:=n+1
```

Program demonstrativ

În continuare vom determina elementele prime dintr-un vector de numere întregi considerate, din start, pozitive. Pentru aceasta vom face o combinație între programul de determinare a primalității unui număr întreg, în general, şi tehnicile de lucru cu vectori, prezentate mai sus.

```
program NumerePrimeDintr_UnVector;
var x: array[1..20] of Integer; rad,i,n,d: Integer; prim: Boolean;
begin
     Write('Dati n = ');  ReadLn(n);
     for i:=1 to n do
         begin Write('Dati x[',i,'] = ');  ReadLn(x[i]) end;
     for i:=1 to n do
         begin { verifica daca x[i] este prim sau nu }
           if x[i]=1 then prim:=False
           else if x[i] mod 2 = 0 then
                   if x[i]=2 then prim:=True
                   else prim:=False
                else
                    begin
                      d:=3;
                      prim:=True;
                      rad:=Trunc(Sqrt(x[i]));
                       while (d<=rad) and prim do
                          if x[i] mod d = 0 then prim:=False
                          else d:=d+2;
                    end;
  { daca numarul x[i] este prim il afisam si afisam si pozitia sa }
           if prim then
              WriteLn('Am gasit nr. prim ', x[i], ' pe pozitia ',i)
         end; ReadLn
end.
```

Exerciții

1. Fie declarațiile: `var a: array[1..20] of Real; n: Integer; e: Real;`
Să se determine valoarea expresiei e în fiecare din cazurile:

a) $e = x_1+x_2+...+x_n$ (adică x[1]+x[2]+...+x[n]);
b) $e = x_1+x_3+...+x_{..}$;
c) $e = x_1 \cdot x_2 \cdot x_3 \cdot ... \cdot x_n$;
d) $e = x_1 \cdot x_3 \cdot x_5 \cdot ... \cdot x_{..}$;
e) e = media aritmetică a componentelor din vector;
f) e = suma pătratelor componentelor din vector;
g) e = suma cuburilor componentelor negative din vector;
h) $e = x_1-x_2+x_3-x_4+... \pm x_n$;

2. Fie doi vectori x şi y, din care doar primele n componente se folosesc:

```
var x,y: array[1..20] of Real; n: Integer;
```

Să se determine vectorul `z: array[1..20] of Real`, astfel încât:

a) z[i]=x[i]+y[i]; { ***suma celor doi vectori*** }
b) z[i]=x[i]-y[i]; { ***diferen]a celor doi vectori*** }
c) z[i]=minimul dintre x[i] şi y[i].

3. Fie doi vectori x şi y, din care doar primele n componente se folosesc:

```
var x,y: array[1..20] of Real; n: Integer;
```

Să se determine expresia e calculată astfel:

a) $e=(x_1+y_1) \cdot (x_2+y_2) \cdot ... \cdot (x_n+y_n)$;
b) $e=x_1 \cdot y_1+x_2 \cdot y_2+...+x_n \cdot y_n$; {***produsul scalar a doi vectori***}
c) $e = minim(x_1,y_n)+minim(x_2,y_{n-1})+minim(x_3,y_{n-2})+...minim(x_n,y_1)$;
d) $e = x_1^2 \cdot y_1+x_2^2 \cdot y_2+...+x_n^2 \cdot y_n$.

Observație: Punctul (b) al acestui exercițiu ne dă prilejul să explicăm de ce se foloseşte în informatică, ca şi în matematică, noțiunea de *vector* pentru a denumi un şir de numere. Vom explica acest lucru, făcând o analogie cu denumirea de vector din fizică. În fizică un vector are forma $\vec{v} = a\vec{i} + b\vec{j} + c\vec{k}$ (în spațiul tridimensional), unde am pus în evidență cei trei versori, deci putem să luăm coeficienții acestor versori şi să-i punem într-un vector din informatică `x=(a,b,c)`, deci `x[1]=a, x[2]=b` şi `x[3]=c`. Produsul scalar a doi vectori din fizică va fi acelaşi ca şi în matematică sau informatică: suma produselor componentelor corespunzătoare: $a_1 \cdot a_2 + b_1 \cdot b_2 + c_1 \cdot c_2$, adică exact ca şi la punctul (b) din problemă. Fireşte, în informatică, spre deosebire de fizică, putem avea vectori mai lungi, iar componentele nu trebuie să fie neapărat numere!

4. Memorați în primele `n` componente ale unui vector `x` de numere întregi primele `n` numere prime.
5. Memorați în primele `n` componente ale unui vector `x` de numere întregi primele `n` numere prime mai mari decit `999`, care citite invers sunt tot numere prime.
6. Fie un vector `x` de numere întregi. Să se formeze un vector `y` de numere întregi, în care `y[i]` să fie reprezentarea în baza `2` a numărului `x[i]`.
7. Fie un vector `x` de numere întregi. Să se formeze un vector `y` de numere întregi, în care `y[i]` să fie restul împarțirii lui `x[i]` la suma cifrelor lui `x[i]`. Complicați exercițiul făcând împărțirea prin scăderi repetate!
8. Fie un vector `x` de numere întregi. Să se determine un număr `p`, care să fie cel mai mare număr prim din cadrul vectorului. Daca nu există, atunci `p` să fie egal cu `0`. Dacă `p` nu este `0`, atunci să se împartă (ca numere întregi) toate componentele lui `x` la suma cifrelor lui `p`. Complicați exercițiul, încercând să determinați radicalul prin metoda lui Newton, iar împărțirea să o faceți prin scăderi repetate!

Tehnica GREEDY
O serie din problemele anterioare au o formulare de genul: să se afişeze toate elementele din vectorul `x` care verifică o anumită relație (de pildă să fie pare şi pe poziții impare, sau să fie prime sau pozitive etc.). Dacă notăm prin `A` mul]imea celor `n` elemente, aceste probleme cer, de fapt, să se găsească o submulțime `B` a lui `A`, care să îndeplinească anumite condiții sau chiar un anumit criteriu de optim (de exemplu, *să se determine elementul maxim care...*). Condiția prezentată este necesară, nu şi suficientă.

Metoda generală de rezolvare a acestor probleme recurgea la parcurgerea "*lacomă*" a vectorului şi ea se numeşte **metoda** sau **tehnica "greedy"**.

Mecanismul metodei "greedy" constă în inițializarea lui B la mulțimea vidă, după care, în mod repetat, până la determinarea întregii mulțimi `B`,:

- *se alege* un anumit element din `A`;
- se testează dacă elementul ales poate fi adăugat mulțimii `B`.

De multe ori mulțimea `A` are un număr foarte mare de elemente, aşa încât alegerea nu se poate face la întâmplare. De asemenea, trebuie văzut dacă elementul ales va fi pus în mulțimea `B` sau nu. Pentru rezolvarea acestor probleme, avem următoarele posibilități:

- dispunem de sau descoperim noi înşine criterii matematice care conduc la rezultatul dorit;
- nu dispunem de astfel de criterii, dar dispunem de altele care conduc la un rezultat acceptabil (nu neapărat optim). În acest caz trebuie să cunoaştem în ce măsură ne-am apropiat de soluția optimă. (Spunem că avem de a face cu o *tehnică euristică*.).

Din cele de mai sus se trage concluzia că această tehnică vă este familiară, chiar dacă nu ați studiat toate tehnicile de programare.

3. Ordonarea şi interclasarea vectorilor

Ordonarea unui vector

Problema cu care am început această carte revine în actualitate, dar sub o formă mai generală: *să se ordoneze crescător componentele unui vector (considerând că acestea se pot compara între ele).* Să presupunem, aşadar, că avem un vector de numere întregi, pe care vrem să le punem în ordine crescătoare; dar în loc de numere, am putea avea şi altceva, de pildă numele unor elevi, pe care dorim să-i punem în ordine alfabetică. În lecția 1 rezolvarea problemei, deşi reprezentată de o procedură recursivă, era următoarea: să îl punem pe cel mai mic element în față, apoi să procedăm la fel cu restul. Pentru aceasta, vom compara primul element `x[1]` cu toate elementele de după el, deci cu elementele de pe pozițiile `2,3,...,n`. Dacă găsim un element `x[j]` care e mai mic decât `x[1]`, atunci îl schimbăm pe `x[1]` cu acel element. Când am ajuns la ultimul element, deci la `x[n]`, înseamnă că pe prima poziție va fi sigur cel mai mic element din vector. Va trebui ca, în continuare, să ne ocupăm doar de elementele de la poziția a doua încolo. De aceea, la fel ca şi în prima trecere prin vector, vom compara elementul `x[2]` cu toate elementele de după el, urmând ca atunci când găsim un element `x[j]` mai mic decât `x[2]`, să-l schimbăm cu acesta şi, deci, să-l aducem pe cea de a doua poziție. Aceste treceri (care ar putea cuprinde eventual interschimbări ale elementului din față cu acel element curent (de pe poziția `j`) de după elementul din față) au loc până la penultima pozi]ie, când ar mai avea loc doar o singură comparare, între `x[n-1]` şi `x[n]`.

Aşadar, pentru a ordona vectorul `x` de `n` elemente, vom proceda în felul următor:

```
pentru i de la 1 la n-1 execută
   pentru j de la i+1 la n execută
         dacă x[j] < x[i] atunci schimbă pe x[i] cu x[j].
```

Programul de mai jos ordonează alfabetic numele unor elevi:

```
program OrdonareAlfabetica;
var x: array[1..20] of String;i,j,n: Integer; aux: String;
begin { citirea datelor }
     Write('Câti elevi sunt ? : '); ReadLn(n);
     for i:=1 to n do
         begin Write(' - nume elev nr. ',i,' : '); ReadLn(x[i]) end;
     for i:=1 to n-1 do { ordonarea vectorului }
         for j:=i+1 to n do
             if x[j]<x[i] then
                begin { schimbare } aux:=x[i]; x[i]:=x[j]; x[j]:=aux end;
     { afisarea rezultatului }
     WriteLn('Ordinea alfabetica este:');
     for i:=1 to n do WriteLn(i,'. ',x[i]);
     ReadLn
end.
```

Metoda pe care am folosit-o mai înainte pentru a ordona un vector este denumită **metoda selecției directe**. Când numărul de componente ale vectorului (`n`) este foarte mare, putem spune că ea necesită n^2 operații (din cauza celor două cicluri `for` imbricate). Există multe metode de sortare, pe care le vom studia mai târziu, unele mai avansate decât altele. În continuare, vom prezenta o metodă la fel de simplă şi de cunoscută ca şi cea dinainte. Este vorba de **metoda "bubble-sort" (bulelor)** , care însă, în general, nu este mai rapidă: ea constă în a compara fiecare element `x[i]` cu elementul de pe poziția succesoare, deci cu elementul `x[i+1]`. Dacă ele nu sunt puse în ordine crescătoare, se vor schimba între ele. Procesul se repetă până când, la o parcurgere a vectorului, nu mai are loc nici o inteschimbare. Acest lucru este indicat de o variabilă booleană `gata`.

```
program SortarePrinMetodaBulelor;
var x: array[1..20] of Integer; n,i,aux: Integer; gata: Boolean;
begin
     Write('Dati nr. de componente : '); ReadLn(n);
     for i:=1 to n do
```

```
          begin Write('Dati x[',i,']='); ReadLn(x[i]) end;
     { ordonarea }
     repeat
        gata:=True;
        for i:=1 to n-1 do
            if x[i+1]<x[i] then
               begin  { interschimbarea are loc }
                 gata:=False; aux:=x[i]; x[i]:=x[i+1]; x[i+1]:=aux
               end
     until gata;
     { afisarea vectorului ordonat }
     WriteLn('Vectorul sortat: ');
     for i:=1 to n do WriteLn('Pe pozitia ',i,' este : ',x[i]);
     ReadLn
end.
```

Aplicație practică: concurs de admitere în liceu

Ordonarea unui vector este un lucru foarte important în programare, iar una dintre cele mai întâlnite situații este dată de ordonarea unor elevi în funcție de mediile obținute de aceştia într-un concurs. Pentru a rezolva această problemă, trebuie să memorăm numele elevilor într-un vector de şiruri de caractere (pe care l-am numit sugestiv `nume`). Mediile vor fi memorate într-un vector de numere reale (fie acesta `media`). Vom citi datele despre fiecare elev în parte, apoi, comparându-le doar mediile, vom interschimba, dacă este cazul, atât mediile, cât şi numele corespunzătoare. În final vom afişa ordonați elevii, după medie.

```
program Concurs;
const max=50;
var nume: array[1..max] of String; media: array[1..max] of Real;
    i,j,NrElevi: 1..max; aux1: String; aux2: Real;
begin
     Write('Dati numarul de elevi : '); ReadLn(NrElevi);
     for i:=1 to NrElevi do
         begin
           WriteLn('Dati datele elevului nr. ',i);
           Write(' - nume si prenume : '); ReadLn(nume[i]);
           Write(' - medie obtinuta  : '); ReadLn(media[i])
         end;
     { se ordoneaza elevii DESCRESCATOR dupa medie }
     for i:=1 to NrElevi-1 do
         for j:=i+1 to NrElevi do
             if media[j]>media[i] then
                begin
                  aux1:=nume[i]; nume[i]:=nume[j]; nume[j]:=aux1;
                  aux2:=media[i]; media[i]:=media[j]; media[j]:=aux2
                end;
     { se afiseaza elevii in ordinea descrescatoare a mediilor }
     WriteLn('Rezultat:');
     for i:=1 to NrElevi do
         WriteLn(i,'. ',nume[i]:30,' : ',media[i]:5:2);
     ReadLn
end.
```

Există o mică problemă nerezolvată de program. Acesta citeşte mediile gata calculate, însă ar fi bine să dăm calculatorului cele două note obținute la probele de concurs (de pildă *limba şi literatura română* şi *matematică*), iar el să ne calculeze mediile şi să ne afişeze rezultatul. De asemenea, e necesar uneori, să avem elevii aranjați în ordinea lor alfabetică, sau după o anumită notă. Un astfel de program este prezentat în continuare, deşi ar fi de dorit ca cititorul să încerce să-l realizeze singur. Programul are şi o interfață prietenoasă, care constă dintr-un meniu color, în care opțiunile se selectează cu ajutorul tastelor de cursor sus şi jos şi a tastei `Enter`. Fiecare opțiune din meniu (memorată într-o componentă a vectorului `meniu`) are asociată o explicație ce va fi afişată în partea de jos a ecranului. Programul permite să se schimbe datele elevilor, prin reintroducerea lor. Un ciclu `repeat-until` controlează execuția

programului, care se încheie atunci când se selectează cea de a şasea opțiune. Structura programului este simplu de urmărit, deoarece se bazează pe o instrucțiune `case` şi o serie de operații cu vectori (citiri, scrieri, ordonări) pe care le-am prezentat deja. Dacă utilizatorul programului va încerca să ordoneze elevii, înainte de a introduce datele asociate lor, atunci programul va semnala eroare. În acest sens, se foloseşte o variabilă logică. Modul cum a fost realizat programul este uşor de înțeles, dacă se încearcă o rulare a sa pas cu pas (cu tastele `F7`, `F8` şi `F4`).

Ceea ce trebuie remarcat în partea de declarații este că se utilizează nişte constante de tip caracter, date prin codurile tastelor `sus`, `jos` şi `Enter`. Astfel, de pildă, `Enter` are codul `13`, de aceea se poate scrie `Enter=#13`. Problema se complică la tastele de cursor, cărora le sunt asociate două coduri, primul fiind `0`, iar al doilea, cel care contează cu adevărat, este `72`, `75`, `77` sau `80`. Vă dorim distracție plăcută!

```
program EleviLaConcursDeAdmitere;
uses Crt;
const max=20; Enter=#13; Sus=#72; Jos=#80;
var nota1, nota2: array[1..max] of Integer;
    media: array[1..max] of Real; nume: array[1..max] of String;
    aux_i,n,i,j,optiune: Integer; aux_r: Real; aux_s: String;
    meniu,explicatie: array[1..6] of String; tasta: Char;
    date_introduse: Boolean;
begin
{ partea de initializare a meniului si a explicatiilor asociate optiunilor }
  meniu[1]:=' Introducere date  '; explicatie[1]:='Se introduc numele elevilor
si cele doua note pentru fiecare  ';
  meniu[2]:='Listare dupa nota 1'; explicatie[2]:='Se afiseaza elevii in ordinea
descrescatoare a primei note    ';
  meniu[3]:='Listare dupa nota 2'; explicatie[3]:='Se afiseaza elevii in ordinea
descrescatoare a note a doua    ';
  meniu[4]:='Listare dupa medie '; explicatie[4]:='Se afiseaza elevii in ordinea
descrescatoare a mediei obtinute';
  meniu[5]:='Listare alfabetica '; explicatie[5]:='Se afiseaza elevii in ordinea
alfabetica a numelor lor        ';
  meniu[6]:='  Oprire program   '; explicatie[6]:='Se opreste programul de
ordonare a elevilor                  ';
  { variabila de control a introducerii datelor despre elevii candidati }
  date_introduse:=False; ClrScr;
  repeat { urmeaza partea de interfata, de selectare a optiunii }
   TextBackground(Blue); ClrScr; TextColor(White+Blink); GoToXY(33,3);
   Write('O R D O N A R E'); TextBackground(Cyan); TextColor(Brown);
   { se afiseaza optiunile meniului, una sub alta }
   for i:=1 to 6 do begin GoToXY(32,8+2*i);Write(meniu[i]) end;
   { prima optiune este cea implicita }
   optiune:=1; GoToXY(32,8+2*optiune);
   TextBackground(LightGreen);Write(meniu[optiune]); TextBackground(Blue);
   TextColor(Yellow); GoToXY(10,24);Write(explicatie[optiune]);
   repeat { se apasa sus, jos iar la sfârsit Enter }
     tasta:=ReadKey; TextColor(Brown); TextBackground(Cyan);
     GoToXY(32,8+2*optiune);Write(meniu[optiune]);
     case tasta of
       Sus: begin optiune:=optiune-1; if optiune=0 then optiune:=6 end;
       Jos: begin optiune:=optiune+1; if optiune=7 then optiune:=1 end
     end;
     GoToXY(32,8+2*optiune); TextBackground(Green); Write(meniu[optiune]);
     TextBackground(Blue); TextColor(Yellow);
     GoToXY(10,24); Write(explicatie[optiune])
   until tasta=Enter;
   ClrScr; TextColor(Yellow);
   { urmeaza programul propriu-zis }
   case optiune of
     1: begin { introducere date elevi }
         WriteLn('INTRODUCERE DATE ELEVI');
         Write('Dati numarul de elevi : '); ReadLn(n);
```

```
     for i:=1 to n do
       begin
        Write('Dati numele elevului nr. ',i,' : '); ReadLn(nume[i]);
        Write('Dati nota 1 a elevului nr. ',i,' : '); ReadLn(nota1[i]);
        Write('Dati nota 2 a elevului nr. ',i,' : '); ReadLn(nota2[i]);
        media[i]:=(nota1[i]+nota2[i])/2
       end;
     date_introduse:=True
    end;

2: if date_introduse then
     begin  { listare dupa nota 1}
      for i:=1 to n-1 do
       for j:=i+1 to n do
        if nota1[i]<nota1[j] then
           begin
            aux_i:=nota1[i]; nota1[i]:=nota1[j]; nota1[j]:=aux_i;
            aux_i:=nota2[i]; nota2[i]:=nota2[j]; nota2[j]:=aux_i;
            aux_s:=nume[i]; nume[i]:=nume[j]; nume[j]:=aux_s;
            aux_r:=media[i]; media[i]:=media[j]; media[j]:=aux_r
           end;
      WriteLn('LISTARE DUPA PRIMA NOTA ');
      for i:=1 to n do WriteLn(i,'. ',nume[i],' : ',nota1[i]); ReadLn
     end
    else begin WriteLn('Nu s-au introdus datele !'); ReadLn end;
3: if date_introduse then
     begin  { listare dupa nota 2}
      for i:=1 to n-1 do
       for j:=i+1 to n do
        if nota2[i]<nota2[j] then
           begin
            aux_i:=nota1[i]; nota1[i]:=nota1[j]; nota1[j]:=aux_i;
            aux_i:=nota2[i]; nota2[i]:=nota2[j]; nota2[j]:=aux_i;
            aux_s:=nume[i]; nume[i]:=nume[j]; nume[j]:=aux_s;
            aux_r:=media[i]; media[i]:=media[j]; media[j]:=aux_r
           end;
      WriteLn('LISTARE DUPA A DOUA NOTA ');
      for i:=1 to n do WriteLn(i,'. ',nume[i],' : ',nota2[i]); ReadLn
     end
    else begin WriteLn('Nu s-au introdus datele !'); ReadLn end;
4: if date_introduse then
     begin  { listare dupa medie}
      for i:=1 to n-1 do
       for j:=i+1 to n do
        if media[i]<media[j] then
           begin
             aux_i:=nota1[i]; nota1[i]:=nota1[j]; nota1[j]:=aux_i;
             aux_i:=nota2[i]; nota2[i]:=nota2[j]; nota2[j]:=aux_i;
             aux_s:=nume[i]; nume[i]:=nume[j]; nume[j]:=aux_s;
             aux_r:=media[i]; media[i]:=media[j]; media[j]:=aux_r
           end;
     WriteLn('LISTARE DUPA MEDIA NOTELOR ');
     for i:=1 to n do WriteLn(i,'. ',nume[i],' : ',media[i]:5:2); ReadLn
    end
   else begin WriteLn('Nu s-au introdus datele !'); ReadLn end;
5: if date_introduse then
     begin  { listare dupa nume}
      for i:=1 to n-1 do
       for j:=i+1 to n do
        if nume[i]>nume[j] then
           begin
            aux_i:=nota1[i]; nota1[i]:=nota1[j]; nota1[j]:=aux_i;
            aux_i:=nota2[i]; nota2[i]:=nota2[j]; nota2[j]:=aux_i;
            aux_s:=nume[i]; nume[i]:=nume[j]; nume[j]:=aux_s;
            aux_r:=media[i]; media[i]:=media[j]; media[j]:=aux_r
           end;
```

```
         WriteLn('LISTARE DUPA MEDIA NOTELOR ');
         for i:=1 to n do WriteLn(i,'. ',nume[i],' : ',media[i]:5:2); ReadLn
        end
       else begin  WriteLn('Nu s-au introdus datele !');  ReadLn  end
  end
until optiune=6 { optiunea de oprire program }
end.
```

Interclasarea a doi vectori

Să presupunem, în continuare, că au avut loc ordonările descrescătoare ale mediilor pentru două licee unde s-au susținut concursuri de admitere. Inspectoratul Çcolar Jude]ean ar putea fi interesat de situația pe ansamblu a notelor obținute de candidații de la ambele licee, deci ordinea descrescătoare, după medie, a reuniunii celor două mulțimi de candidați. O primă soluție ar fi să realizăm reuniunea mulțimilor de candidați, apoi să o sortăm. Dar ar fi mai bine să ne folosim de faptul că avem deja ordonați elevii pe două liste. De aceea, vom face o **interclasare** a celor două liste.

Astfel, presupunând că dispunem de doi vectori (eventual de dimensiuni diferite), ordonați, să ob]inem vectorul reuniune ordonat. Fie `a` şi `b` cei doi vectori, iar în vectorul `c` vrem să punem rezultatul. Vom proceda în felul următor:

- vom compara *primul* element din vectorul `a` cu primul din `b` şi pe cel mai mic îl vom pune în `c`, eliminându-l din vectorul de unde provine;
- procesul se repetă până când se epuizează unul din vectori;
- apoi se copiază la sfârşitul lui `c` toate elementele din vectorul rămas neterminat, care, fireşte, vor fi şi deja ordonați, cât şi mai mari ca ultimul element din vectorul care s-a terminat primul.

De fapt nu vom face nici o eliminare a unui element din vectorul `a` sau `b`. Vom folosi două variabile întregi, `i` şi respectiv `j`, care să indice mereu pe "*primul*" element din `a`, respectiv din `b`. Apoi, elementul curent adăugat în vectorul `c` va fi pe poziția `k`. Programul este prezentat mai jos şi, după cum observați, algoritmul pe care se bazează este dat de tehnica "greedy".

```
program InterclasareVectori;
const max=10;
var a,b: array[1..max] of Integer; c: array[1..2*max] of Integer;
    m,n,i,j,k: Integer;
begin
     WriteLn('Interclasare vectori deja ordonati');
     Write('Dati nr. de elemente ale vectorului "a" : '); ReadLn(m);
     for i:=1 to m do begin Write('Dati a[',i,'] = '); ReadLn(a[i]) end;
     Write('Dati nr. de elemente ale vectorului "b" :'); ReadLn(n);
     for i:=1 to n do begin Write('Dati b[',i,'] = '); ReadLn(b[i]) end;
     { urmeaza procesul repetitiv de punere a celui mai mic din cele doua
       capete in vectorul rezultat: c }
     i:=1; j:=1; k:=0;
     while (i<=m) and (j<=n) do
           if a[i]<b[j] then
              begin { a[i] ("primul" din a) este minim, se pune in c}
                k:=k+1; c[k]:=a[i]; i:=i+1
                { "eliminarea" lui a[i], adica avansarea in vectorul a}
              end
           else { se pune b[j] in vectorul rezultat c }
              begin k:=k+1; c[k]:=b[j]; j:=j+1 end;
     { acum se adauga elementele din vectorul neterminat }
     if i<=m then while i<=m do begin k:=k+1; c[k]:=a[i]; i:=i+1 end
     else while j<=n do begin k:=k+1; c[k]:=b[j]; j:=j+1 end;
     { afisarea lui c, vector de lungime k }
     WriteLn('Vectorul rezultat este: ');
     for i:=1 to k do Write(c[i]:3,','); ReadLn
end.
```

În acest moment, putem spune că o metodă de sortare eficientă ar fi una care să folosească interclasarea a două jumătă]i deja ordonate ale vectorului. Astfel, dacă dimensiunea vectorului

ce trebuie sortat este destul de "mare" (de pildă mai mare decât 2, căci a sorta două elemente este foarte simplu!), atunci se împarte vectorul în două; se sortează cele două părți (eventual recursiv), apoi se interclasează. Vom reveni asupra acestei metode într-o lecție dedicată, printre altele, şi metodelor de sortare.

4. Alte aplicații ale vectorilor

Histograme

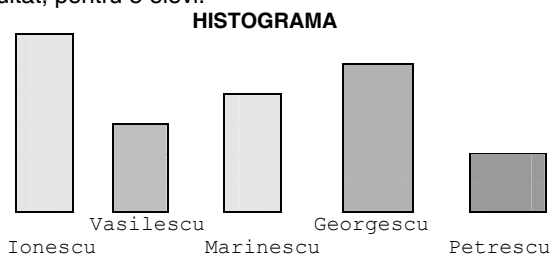
Să presupunem că avem informații despre un număr de maxim 10 elevi (numele şi punctajele lor obținute la un concurs de informatică). Vrem să realizăm o histogramă a rezultatelor lor.

Prin **histogramă** înțelegem o diagramă reprezentînd prin dreptunghiuri (bare verticale) colorate o anumită distribuție statistică.

Fiecare elev va fi trecut cu numele său şi cu o bară verticală, de o anumită culoare, ce reprezintă situația sa, adică punctajul său. Soluția este dată de programul de mai jos, care utilizează biblioteca `Crt`. El poate fi îmbunătățit, pentru a funcționa pentru mai mult de 10 elevi. Fiecare bară dreptunghiulară este de grosime 5, fiind formată din caractere cu codul `ASCII 219`, adică dreptunghiuri umplute. Înălțimile acestor bare (ce pornesc de sus în jos) sunt de două ori mai mari decât punctajul asociat. Numele elevilor sunt scrise în partea de jos a ecranului, pe două linii, una pentru numele de pe poziții impare, iar alta pentru cele de pe poziții pare, pentru a evita eventualele suprapuneri. Fiecare elev al `i`-lea are situația afişată în culoarea `i+1`, bara sa începând pe coloana `7*i-1`. Această formulă se va schimba în cazul în care se doreşte ca programul să permită reprezentarea unui număr mai mare de bare.

```
program Histograma;
uses Crt;
var punctaj: array[1..10] of 1..10; nume: array[1..10] of String;
    NrElevi, i,j: Integer;
begin
     ClrScr; TextColor(LightBlue+Blink);
     GoToXY(30,1); Write('H I S T O G R A M A');
     TextBackground(Blue); TextColor(LightGreen);
     GoToXY(30,3); Write('Nr. elevi (max 10) :'); ReadLn(NrElevi);
     TextBackground(Blue); TextColor(White);
     for i:=1 to NrElevi do
         begin
           GoToXY(25,3+2*i); Write('Nume elevi ',i,' : ');
           ReadLn(nume[i]);
           GoToXY(25,4+2*i); Write('Punctaj elev : '); ReadLn(punctaj[i])
         end;
     ClrScr; TextColor(White);
     GoToXY(30,1); Write('H I S T O G R A M A');
     for i:=1 to NrElevi do
         begin
           TextColor(i+1);
           for j:=1 to 2*punctaj[i] do
               begin GoToXY(7*i,22-j); Write(#219#219#219#219#219) end;
           if odd(i) then GoToXY(7*i-1,24) else GoToXY(7*i-1,23);
           Write(nume[i])
         end; ReadLn
end.
```

Iată un posibil rezultat, pentru 5 elevi:



Operații cu mulțimi, memorate sub formă de vectori

De multe ori - aşa cum se va vedea mai târziu - este necesar să operăm cu mulțimi. Limbajul Pascal permite lucrul cu mulțimi, având chiar posibilitatea de a defini tipuri mulțime, după cum vom vedea mai târziu. Din păcate nu se poate lucra decât cu mulțimi de maxim 255 de elemente, iar acestea trebuie să fie de tip ordinal (scalar). De asemenea controlul programatorului asupra unui element din mulțime este destul de dificil de realizat. De aceea, este util să memorăm mulțimile sub forma unor vectori, făcând, însă, convenția ca să nu se repete elementele în cadrul vectorului.

Să presupunem că avem doi vectori `A` (de `m` elemente distincte) şi `B` (de `n` elemente distincte). Să realizăm reuniunea şi respectiv intersecția lor.

- pentru reuniune, vom copia în vectorul rezultat (`Reun`) toate elementele din `A`, după care vom adăuga acele elemente din `B` care nu se găsesc şi în `A`;
- pentru intersecție, vom pune în mulțimea rezultat (`Inters`) toate elementele din `A` care se regăsesc în `B`.

```
program Reuniune;
{ reuneste multimea A de m elemente cu multimea B de n elemente;
  se obtine multimea Reun cu p elemente }
type multime=array[1..20] of Integer;
var A,B,Reun: multime; m,n,p,i,j: Integer; gasit: Boolean;
begin
 Write('Dati nr. de elemente ale primei multimi  : '); ReadLn(m);
 for i:=1 to m do begin Write('Dati elem. A[',i,']='); ReadLn(A[i]) end;
 Write('Dati nr. de elemente ale multimii a doua : '); ReadLn(n);
 for i:=1 to n do begin Write('Dati elem. B[',i,']='); ReadLn(B[i]) end;
 { se presupune ca A nu are elemente care sa se repete, nici B }
 { se vor copia in Reun toate elementele lui A }
 for i:=1 to m do Reun[i]:=A[i];
 p:=m; { deocamdata Reun are m elemente }
 { apoi se adauga acele elemente din B care nu sunt in A }
 for i:=1 to n do
  begin
   { deci se cauta B[i] in A }
   j:=1; gasit:=False;
   while (j<=m) and (not gasit) do
     if B[i]=A[j] then gasit:=True else j:=j+1;
        if not gasit then
           begin { este un element nou, deci trebuie adaugat }
             p:=p+1; Reun[p]:=B[i]
           end
  end;
  { afisez multimea reuniune }
  WriteLn('Reuniunea multimilor este: ');
  for i:=1 to p do Write(Reun[i]:2,',');
  ReadLn
end.
program Intersectie;
{ intersectaza multimea A de m elemente cu multimea B de n elemente;
  se obtine multimea Inters cu p elemente }
```

```
type multime=array[1..20] of Integer;
var A,B,Inters: multime; m,n,p,i,j: Integer; gasit: Boolean;
begin
 { ..... }
 { ... se citesc cele doua multimi, ca si la reuniune ... }
 { se parcurge A, se pun in Inters toate acele el. ce se regasesc in B }
 p:=0;
 for i:=1 to m do
  begin
   j:=1; gasit:=False;
   while (j<=n) and (not gasit) do
     if A[i]=B[j] then gasit:=True else j:=j+1;
   if gasit then begin { punem in Inters } p:=p+1; Inters[p]:=A[i] end
  end;
 WriteLn('Intersectia multimilor este: '); { afisez multimea inters. }
 for i:=1 to p do Write(Inters[i]:2,','); ReadLn
end.
```

Cele două mulțimi ar putea fi candidații la un concurs de admitere. Se poate verifica, făcând intersecția mulțimilor, dacă nu cumva există un elev care să se fi înscris la concurs la ambele licee, încălcând, astfel, legea!

Operații cu polinoame

O expresie de forma $a_n X^n+a_{n-1} X^{n-1}+...+a_2 X^2+a_1 X+a_0$, în care a_n , a_{n-1} , ..., a_1 , a_0 sunt numere reale date, numite *coeficienți*, se numeşte **polinom** cu coeficienți reali, de grad n. `X` este *variabila* polinomului.

Problema care apare, în primul rând, este de a calcula *valoarea polinomului* pentru o valoare a variabilei `X` dată. Este de bun simț să memorăm un polinom prin coeficienții săi (un vector de numere reale) şi prin numărul `n`:

```
const max=20; var a: array[1..max] of Real; n: Integer;
```

Apoi apar probleme de genul: suma şi diferența a două polinoame (care se reduce la a aduna, respectiv a scădea, componentele corespunzătoare din vectorii în care memorăm cele două polinoame), precum şi înmulțirea sau împărțirea a două polinoame, lucruri la care ne vom referi mai târziu. Deocamdată, să vedem cum se calculează valoarea unui polinom într-un punct dat. În primul rând, este de la sine înțeles că este mai simplu să calculăm recursiv puterea X^i (din deja calculatul X^{i-1}) decât să înmulțim de `i` ori pe `X` cu el însuşi. Aşadar, avem următorul program:

```
program ValoarePolinom_cu_vector;
const max=20;
var a: array[1..max] of Real; n,i: Integer; x,x_la_i,P_de_x: Real;
begin
  Write('Dati gradul polinomului : '); ReadLn(n);
  for i:=n downto 0 do
       begin Write('Dati coef. a',i,' : '); ReadLn(a[i]) end;
  Write('Dati valoarea lui x : '); ReadLn(x);
  P_de_x:=0; x_la_i:=1;
  for i:=0 to n do
      begin P_de_x:=P_de_x+a[i]*x_la_i; x_la_i:=x_la_i*x end;
  WriteLn('Valoarea polinomului este : ',P_de_x:7:4); ReadLn
end.
```

Interesant este faptul că ne putem lipsi de memorarea coeficienților polinomului într-un vector, citindu-i pe măsură ce îi introducem (vom folosi doar o variabilă reală `a`). Acest lucru este dat de fapul că putem exprima polinomul de mai sus, sub forma:

$$P(x) = x\cdot(x\cdot(x\cdot(....\cdot(x\cdot(a_n)+a_{n-1})+.....)+a_2)+a_1)+a_0 .$$

Aşadar, la un pas oarecare `i`, valoarea polinomului se va calcula în funcție de valoarea sa la pasul precedent: `P:=P· x + a`, unde `a` este valoarea coeficientului de la pasul `i`, adică a_i .
Programul este:

```
program ValoarePolinomFaraVector;
var n,i: Integer;  x,a,P: Real;
```

```
begin
  Write('Dati gradul polinomului : '); ReadLn(n);
  Write('Dati valoarea lui x : '); ReadLn(x);
  P:=0;
  for i:=n downto 0 do
      begin Write('Dati coef. a',i,' ');  ReadLn(a); P:=P*x+a end;
  WriteLn('Valoarea polinomului este : ',P:7:4); ReadLn
end.
```

Animație cu bile

Un efect grafic interesant (pentru începutul unui program mai complex) am putea obține prin vizualizarea mişcării unor bile de diferite culori pe ecran. Acestea să se deplaseze ca pe o masă de biliard, dar să se supună legii reflexiei, nu şi a frecării. Vom avea nevoie, astfel de 5 vectori: doi pentru coordonatele bilelor (`X` şi `Y`), unul pentru culorile bilelor (`Culoare`) şi încă doi pentru deplasările relative (pe orizontală şi pe verticală) ale celor `n` bile (`dX` şi `dY`, având componente ce iau doar valorile `-1` şi `+1`.). Restul programului este o "multiplicare" a jocului cu bila pe masa de biliard, prezentat în capitolul precedent:

```
program MaiMulteBile;
uses Crt;
const max=20;
type vector=array[1..max] of Integer;
var X,Y,dX,dY,Culoare: vector; n,i: Integer;
begin
  ClrScr; { initializare atribute bile }
  Write('Numarul de bile = '); ReadLn(n); ClrScr;
  for i:=1 to n do
     begin
       X[i]:=Random(77)+2; Y[i]:=Random(22)+2;
       Culoare[i]:=1+Random(15);
       dX[i]:=Random(2); if dX[i]=0 then dX[i]:=-1;
       dY[i]:=Random(2); if dY[i]=0 then dY[i]:=-1
     end;
   repeat
    for i:=1 to n do
        begin
      { sterge bila i din pozitia veche si afiseaz-o in noua pozitie;
        ai grija la margini ! }
          GoToXY(X[i],Y[i]); Write(' '); X[i]:=X[i]+dX[i];
          Y[i]:=Y[i]+dY[i];
          if (X[i]=1) or (X[i]=80) then
             begin
               Sound(500); Delay(10); NoSound;  dX[i]:=-dX[i]
             end;
          if (Y[i]=1) or (Y[i]=25) then
             begin
               Sound(100*i); Delay(10); NoSound; dY[i]:=-dY[i]
             end;
          TextColor(Culoare[i]); GoToXY(X[i],Y[i]); Write(#15);
        end;
    Delay(50)
   until KeyPressed;
   ClrScr
end.
```

Vom învăța că există şi alte posibilități de memorare a atributelor unei bile: sub forma unui vector, sub forma unei înregistrări sau chiar sub forma unui obiect. Astfel, această "problemă a bilelor" va fi reluată de mai multe ori în cadrul cărții, pentru a prezenta diferitele feluri de memorare a informațiilor ataşate bilelor, folosind diferite structuri de date.

5. Tipul `String`

Funcții şi proceduri

Revenim asupra tipului de date `String`, datorită asemănării sale foarte mari cu un vector de caractere. O variabilă de tip `String` va avea valoarea unei înşiruiri de caractere, încadrate de apostrofuri.

Astfel, dacă avem `var S: String`, e ca şi cum am dispune de un vector de 255 de caractere. Putem să limităm această mărime la `m` caractere prin `String[m]`. Ca şi în cazul vectorilor, prin `S[i]` vom înțelege caracterul de pe pozi]ia `i` din şirul `S`. Însă, spre deosebire de vectori, un şir `S` poate fi atât citit cât şi afişat în întregime.

- De asemenea, nu este nevoie să se citească lungimea sa, deoarece, după un apel `ReadLn(S)` lungimea sa este cunoscută (determinată de numărul de caractere introduse). Chiar şi în urma unei atribuiri de forma `S:='abc'` se fixează lungimea şirului (aici 3), iar această lungime poate fi aflată folosind funcția **`Length`**. Astfel, `WriteLn(Length(S))` va afişa `3`, pentru şirul de mai înainte.
- Operația de concatenare (alipire) a şirurilor, cunoscută şi până acum, se realizează fie prin operatorul "+", fie cu ajutorul funcției **`Concat`**, care poate avea mai multe argumente, adică, de exemplu:

```
var S1, S2, S, T : String;
S1:='abc'; S2:='def'; S:=Concat(S1,S2); T:=Concat(S,S2,S1)
```

În urma celor 4 atribuiri de mai sus, vom avea şirurile:

```
S='abc', S2='def', S='abcdef', T='abcdefdefabc'.
```

- De multe ori vom avea nevoie să extragem anumite părți dintr-un şir, adică un **subşir**. Astfel, dacă `S` este o variabilă de tip `String`, iar `T` este o expresie de acelaşi tip, putem extrage în `S` subşirul din `T` începând cu poziția `p` şi având lungimea `L`, astfel:

```
S:=Copy(T,p,L).
```

Dacă, începând cu poziția `p` şirul are mai puțin de `L` caractere, se vor extrage toate până la capăt. De pildă, prin `S:=Copy('covorasul',4,3)`, `S` devine şirul `'ora'`.

- Prima poziție a unui subşir într-un şir, adică poziția în şirul mare a primei litere din cadrul subşirului, se determină folosind funcția **`Pos`**. Astfel, prin `p := Pos(S,T)` obținem în variabila `p` de tip întreg (`byte`), prima apariție a lui `S` în `T`. Dacă nu există, `p` va fi `0`.

De pildă, `Pos('bcd','abcdefg')` returnează `2`, iar `Pos('ma','mamaliga')` returnează `1` (deşi `'ma'` apare şi pe poziția `3`). De asemenea, `Pos('abc','defghijkl')` returnează `0`.

- Dacă se doreşte inserarea unui subşir `S` într-un şir `T` înaintea caracterului de pe poziția `p`, atunci se va folosi un apel al procedurii **`Insert`**, de genul: `Insert(S,T,p)`.
- Ştergerea unui subşir de lungime `L` dintr-un şir `S`, începând cu poziția `p` se realizează cu procedura **`Delete`**, prin: `Delete(S,p,L)`.
- În unele aplicații mai complexe va trebui să convertim şiruri reprezentând numere în numerele respective. De multe ori avem nevoie de conversia inversă, din număr în şir de caractere. Aceste lucruri sunt realizate de procedurile **`Val`**, respectiv **`Str`**.

Astfel, dacă vrem să evaluăm într-o variabilă numerică `v` (de tip `Integer` sau `Real`) un şir de caractere `S`, vom scrie `Val(S,v,cod)`, unde `cod` este o variabilă întreagă reprezentând poziția caracterului din şirul `S` unde conversia nu s-a mai putut realiza (din cauza unui caracter nepermis). Astfel, după `Val('123',v,cod)`, `v` obține valoarea `123`, iar `cod=0`, deoarece conversia a reuşit, nicăieri nesemnalându-se eroare. De asemenea, după apelul `Val('123.45',v,cod)`, avem `v=123.45`, iar `cod=0`. Dacă, însă, încercăm ceva de genul: `Val('123#45',v,cod)`, se obține `v=0`, iar `cod=4`, pentru că simbolul nepermis '`#`' apare pe cea de a patra poziție.

Invers, dacă v este o valoare reală, putem să o convertim la un şir de caractere S, (chiar şi cu m poziții, din care n zecimale,) printr-un apel de forma: Str(v:m:n, S). Dacă, însă, v este de tip real sau întreg, prin Str(v:m,S) obținem conversia la un şir S de m caractere. Când v este doar de tip întreg, putem converti, simplu, prin Str(v,S).

- În continuare, să reamintim că există o listă a codurilor ASCII, asociate tuturor caracterelor. Astfel, limbajul Pascal ne pune la dispozi]ie două funcții ce permit "conversia" din caracter într-un număr întreg (egal cu codul său ASCII):

Chr(cod) = caracterul având codul cod, de exemplu Chr(65)='A', Chr(97)='a';
Ord(car) = codul ASCII al caracterului car, de exemplu, Ord('A')=65, Ord('a')=97.

Pentru a transforma dintr-o literă mică într-o literă mare, se poate scrie astfel:
LitMare:=Chr(Ord(LitMica)-32), dar se poate folosi funcția predefinită UpCase:
LitMare:=UpCase(LitMica); funcția nu afectează celelalte caractere.
Invers, conversia se va realiza astfel: LitMica := Chr(Ord(LitMare)+32), neexistând o funcție predefinită pentru această conversie.

Exemple de utilizare

1. Transformări din litere mici în litere mari şi invers
Secvența de program care transformă un şir scris cu litere mici în acelaşi şir, cu litere mari este: for i:=1 to Length(S) do S[i]:=UpCase(S[i]).
Conversia inversă, realizată cu funcțiile Chr şi Ord va căuta să nu afecteze celelalte caractere:

```
for i:=1 to Length(S) do
    if (S[i]>='A') and (S[i]<='Z') then S[i]:=Chr(Ord(S[i]+32) .
```

2. Limba păsărească
Problema - în varianta de aici - cere înlocuirea fiecărei vocale V dintr-un şir prin grupul pVp. Astfel, vom parcurge şirul şi, ori de câte ori vom întâlni o vocală, pe o anumită poziție i, vom insera înaintea ei o literă "p" şi la fel după ea. În fine, vom trece în şir după acest ultim "p".

```
program Pasareasca;
var S: String; i: Integer;
begin
  Write('S='); ReadLn(S);
  i:=1;
  while i<=Length(S) do
    if (S[i]='a') or (S[i]='e') or (S[i]='i') or
       (S[i]='y') or (S[i]='o') or (S[i]='u') then
      begin Insert('p',S,i); i:=i+2;  Insert('p',S,i); i:=i+1 end
    else i:=i+1;
  WriteLn('S final = ',S);
  ReadLn
end.
```

3. Despărțirea unei propoziții în cuvinte şi numărarea lor
Această problemă prezintă interes în multe aplicații ce țin de editoarele de texte sau de prelucrarea unui anumit text, scris într-un limbaj de programare sau chiar într-o anumită limbă. Fie S un şir de caractere. Considerăm că el este o succesiune de cuvinte despărțite prin spații. Vom memora aceste cuvinte într-un vector numit Cuv:
var S: String; Cuv: array[1..20] of String; NrCuv: Integer;
Numărul acestor cuvinte va fi memorat în NrCuv. Cum vom proceda? Vom depista poziția primului spațiu în cadrul lui S, fie aceasta p. Preluăm drept prim cuvânt, acea porțiune din S, de la început până la pozi]ia p-1. Apoi ştergem din S până la poziția p inclusiv. Procedăm tot aşa şi mai departe, până se epuizează şirul. Pentru a putea

prelua şi ultimul cuvânt după acelaşi principiu şi pentru a asigura funcționalitatea procedurii, va trebui ca la început să adăugăm un spațiu la sfârşitul şirului `S`.

```
program DespartireInCuvinte;
var S: String; Cuv: array[1..20] of String; p,i,NrCuv: Integer;
begin
 Write('Dati propozitia : '); ReadLn(S); S:=S+' '; NrCuv:=0;
 while S<>'' do
  begin
   p:=Pos(' ',S);Inc(NrCuv);Cuv[NrCuv]:=Copy(S,1,p-1);Delete(S,1,p)
  end;
 WriteLn('Cuvintele sale sunt:');
 for i:=1 to NrCuv do WriteLn(i,': ',Cuv[i]); ReadLn
end.
```

În cele ce urmează vom încerca să îmbunătățim acest program în vederea punerii în vectorul `Cuv` a cuvintelor distincte ce apar în şir. De asemenea, vom memora în vectorul `Ap` numerele de apariții ale cuvintelor din `Cuv`. Vom determina, ca şi până acum, cuvântul curent. Acesta va fi pus în vectorul `Cuv` numai dacă nu exista deja, caz în care se incrementează `NrCuv` şi se pune `Ap[NrCuv] = 1`. Dacă cuvântul exista deja, atunci el nu se mai adaugă în vector, însă se incrementează numărul său de apariții; astfel, dacă el se află deja în vector, pe o poziție `j`, atunci vom scrie: `Ap[j]:=Ap[j]+1`.

```
program ContorizareCuvinte;
var S, cuvint: String; Cuv: array[1..20] of String[20];
    Ap:array[1..20] of Integer; p,j,NrCuv: Integer; gasit:Boolean;
begin
  Write('Dati propozitia : '); ReadLn(S); S:=S+' '; NrCuv:=0;
  while S<>'' do
    begin
     p:=Pos(' ',S); cuvint:=Copy(S,1,p-1); j:=1; gasit:=False;
     while (j<=NrCuv) and (not gasit) do
           if cuvint = Cuv[j] then gasit:=True else Inc(j);
     if not gasit then
        begin Inc(NrCuv); Cuv[NrCuv]:=cuvint; Ap[NrCuv]:=1 end
     else Ap[j]:=Ap[j]+1;
     Delete(S,1,p)
    end;
  WriteLn('Cuvintele sale sunt:');
  for j:=1 to NrCuv do
      WriteLn(j,': ',Cuv[j], ' - apare de ',Ap[j], ' ori');
  ReadLn
end.
```

4. Afişarea unui şir prin litere care cad

Jocurile pe calculator, pe care le puteți realiza şi dumneavoastră, cu cunoştințele de până acum, pot avea prezentări dintre cele mai frumoase. Un efect interesant şi atractiv îl reprezintă căderea unor litere; ele se depun pe un rând al ecranului, formând un text.

```
program SirInCadere;
uses Crt;
var S: String; i,j: Integer; tasta: Char;
begin
  TextBackground(Black); ClrScr; WriteLn('Efectul de cadere: ');
  Write('Dati sirul : '); ReadLn(S);
  ClrScr; if S='' then S:='PREZENTARE';
  for i:=1 to Length(S) do
      for j:=2 to 20 do
          begin
            GoToXY(10+2*i,j-1); Write(' ');
            GoToXY(10+2*i,j); TextColor(i); Write(S[i]);
            GoToXY(1,1); Sound(200-10*j); Delay(10)
```

```
            end;
        NoSound; tasta:=ReadKey
end.
```

5. Scrisul defilant

Pe linia 20 va apare, defilând de la dreapta la stânga, un mesaj. El apare din extrema dreaptă a ecranului şi intră în cea stângă. Procesul se repetă până se apasă o tastă. Pentru a realiza acest lucru, se foloseşte un mic truc: se pun în fața şi în spatele şirului (mesajului) inițial câte 80 de spații. Apoi se afişează câte 80 de caractere din şir, începând cu o poziție `i` ce creşte de la `1` la `Length(S)-79`. Simplu, nu ?

```
program ScrisMergator;
uses Crt;
var S: String; i: Integer;
begin
     WriteLn('Dati sirul de caractere (maxim 95 caractere) !');
     ReadLn(S); for i:=1 to 80 do S:=' '+S+' '; ClrScr;
     repeat
       for i:=1 to Length(S)-79 do
           begin GoToXY(1,20); Write(Copy(S,i,80)); Delay(50) end
     until KeyPressed
end.
```

6. Tablouri bidimensionale (matrice)

Declarație. Exemple

Să presupunem că avem un şir de `n` elevi, $n \leq 20$, fiecare din ei având un nume şi un prenume. Pentru a reține aceste date despre ei, utilizând structurile de date învățate până acum, vom folosi două tablouri unidimensionale:

```
var nume, prenume: array[1..20] of String;
```

Prin `nume[i]` se înțelege numele elevului al `i`-lea, iar prin `prenume[i]` prenumele.

Putem, însă, să considerăm un tablou unidimensional de vectori cu două componente, pentru a memora atât numele, cât şi prenumele elevilor:

```
var elev: array[1..20] of array[1..2] of String;
```

Astfel, prin `elev[i][1]` vom înțelege numele elevului al `i`-lea, iar prin `elev[i][2]` prenumele acestuia.

Limbajul Pascal ne permite, însă, să definim tablouri dimensionale, astfel încât, variabila `elev` de mai sus să poată fi declarată sub forma:

```
var elev: array[1..20, 1..2] of String;
```

Este ca şi cum am avea într-adevăr un tabel, cu 20 de linii şi două coloane, pe fiecare linie fiind, în cele două coloane, numele şi respectiv prenumele unui elev:

- Declarația generală a unui tablou bidimensional (adică având şi linii şi coloane) este:

```
type id_tip_tablou_bidimens = array[id_tip_sc_1, id_tip_sc_2] of  id_tip
```

Deci se defineşte un tip nou de date (`id_tip_tablou_bidimens`) ca fiind un tablou bidimensional, cu indicii în domeniile scalare `id_tip_sc_1` şi `id_tip_sc_2`, tablou ale cărui elemente sunt de tipul `id_tip`. Un tablou bidimensional se mai numeşte şi **matrice**, conform noțiunii din matematică. În mod similar, se pot defini şi tablouri cu mai mult de două dimensiuni.

Exemple:

- Să presupunem că vrem să facem un program pentru un orar pentru elevii din clasele I-IV. Vom defini o matrice în care, pe coloane vom avea zilele săptămânii, iar pe linii orele de lucru. Drept elemente constituente vor fi nişte şiruri de caractere, reprezentând disciplinele de învățământ.

```
type orar=array[1..4,1..5] of String;
var OrarulMeu: orar;
```

	1	2	3	4	5
1	Citirea	Caligrafia	Citirea	Matematică	Matematică
2	Matematica	Citirea	Matematică	Desen	Cunoşt inte despre natură
3	Ed. fizică	Cunoşt. despre natură	Lucru manual	Religie	Citirea
4	Lb. engleză	Ed. fizică	Desen	Citire	Lucru manual

Putem, de asemenea, să definim direct:

```
var OrarulMeu: array[1..4,1..5] of String; .
```

- Dar, pentru a fi mai sugestivi, am putea face declarații de genul:

```
type zi_de_scoala=(luni, marti, miercuri, joi, vineri);
     orar = array[1..4, zi_de_scoala] of String;
```

- Să definim o matrice pentru o tablă de şah; piesele vor fi codificate: `-1` pion negru, `+1` pion alb, `-2` cal negru, `+2` cal alb, ş.a.m.d., iar `0` pentru pătrățele neocupate.

```
type tabla_de_sah = array['a'..'h', 1..8] of Integer;
```

- Să presupunem că avem un grup de `n` băieți şi `n` fete care sunt prietene cu cei `n` băie]i.

Vom memora aceasta sub forma unei matrice *pătratice* (are numărul de coloane egal cu cel de linii):

```
type matrice = array[1..20, 1..20] of Boolean;
var X: matrice; n: 1..20;
```

Vom avea `X[i,j]=True` dacă băiatul `i` este prieten cu fata `j`, respectiv `False`, dacă nu.

Programe demonstrative:

1. Cel mai bun dintre toți.

Într-o şcoală sunt `m` clase, fiecare având câte `n` elevi. Se cere să se determine care elev are media la învățătură cea mai mare.

Vom memora 'ntr-o matrice toate mediile tuturor elevilor. Astfel, vom considera:

`var Medie: array[1..30, 1..30] of Real`, cu `Medie[i,j]` = media obținută de elevul al `j`-lea din clasa numărul `i` din şcoala respectivă.

Problema se reduce la determinarea celui mai mare element din întreaga matrice. De aceea, se pleacă de la ipoteza că cel mai mare element este primul (cel din colțul stânga sus) şi, până la proba contrarie, el rămâne cel mai mare. Parcurgându-se matricea `Medie`, acest element maxim se poate modifica, actualizându-se la valoarea unui element găsit mai mare. Se păstrează, de fiecare dată, în variabilele `i0` şi `j0` poziția pe linie, respectiv coloană, a elementului maxim. Programul este:

```
program CelMaiBunElev;
uses Crt;
var Medie: array[1..30, 1..30] of Real;
    m,n,clasa,elev,i0,j0: Integer; maxim: Real;
begin
  ClrScr; WriteLn('CEL MAI BUN ELEV LA INVATATURA');
  Write('Dati numarul de clase din scoala: '); ReadLn(m);
  Write('Dati numarul de elevi pe o clasa: '); ReadLn(n);
  WriteLn; WriteLn('Dati mediile !');
  for clasa:=1 to m do
      begin
        WriteLn('Clasa nr. ',clasa, ' - dati mediile !');
        for elev:=1 to n do
            begin
              Write(' elevul ',elev,': media: ');
              ReadLn(Medie[clasa,elev])
            end;
      end;
  maxim:=Medie[1,1]; i0:=1; j0:=1;
  for clasa:=1 to m do
      for elev:=1 to n do
          if Medie[clasa,elev]>maxim then
             begin maxim:=Medie[clasa,elev]; i0:=clasa; j0:=elev end;
```

```
  WriteLn('Unul din cei mai buni elevi este:');
  WriteLn('Elevul nr. ',j0,' din clasa nr. ',i0,' .');
  WriteLn('Media maxima (obtinuta de el) este: ',maxim:5:2);
  ReadLn
end.
```

2. Cel mai fericit băiat.

Problema următoare constă în determinarea unui băiat care este cel mai iubit de către fete. Există `NrB` băieți şi `NrF` fete care îi iubesc, însă fiecare fată iubeşte numai anumiți băieți. Se cere să se determine băiatul pe care îl iubesc cele mai multe fete, precum şi topul tuturor băieților. Vom folosi următoarele tablouri: `NumeB, NumeF`: vectori de şiruri de caractere, reprezentând numele personajelor noastre; `Iubit`: matrice cu valorile `1` (când o fată iubeşte un băiat) şi `0` (când nu); `Punctaj`: vector cu "punctajele" obținute de fiecare băiat.

Astfel, avem `Iubit[i,j]=1` dacă fata `NumeF[j]` îl iubeşte pe băiatul `NumeB[i]`, respectiv `0`, când nu (nu contează deloc dacă băiatul o iubeşte pe fată, această informație este de prisos, neinfluențând rezultatul), iar `Punctaj[i] = Iubit[1,i] + Iubit[2,i] + ... + Iubit[NrF,i]`, adică numărul de fete care îl iubesc pe băiatul `NumeB[i]`.

Vom face o ordonare a liniilor matricei `Iubit`, ordonare care va duce şi la ordonarea vectorilor `Punctaj` şi `NumeB`, dar nu va afecta pe vectorul cu numele fetelor.

La sfârşit, dacă există cel puțin doi băieți cu punctajele egale, atunci nu există un unic "fericit" şi se va afişa un mesaj corespunzător. Iată o variantă de program:

```
program CelMaiFericitBaiat;
uses Crt; const max=20;
var NumeF, NumeB: array[1..max] of String;
    Iubit: array[1..max, 1..max] of Integer;
    Punctaj: array[1..max] of Integer; raspuns: String[2];
    NrB,NrF,i,j,fata: Integer; aux_i: Integer; aux_s: String;
begin
  ClrScr; WriteLn('Intrebare: Care baiat este cel mai fericit ?');
  WriteLn('Raspuns: Cel pe care il iubesc cele mai multe fete !');
  { se citesc numarul de baieti si numele lor }
  WriteLn;Write('Dati numarul de baieti: '); ReadLn(NrB);
  for i:=1 to NrB do
    begin Write('. numele baiatului nr. ',i,' :'); ReadLn(NumeB[i]) end;
  Write('Dati numarul de fete: '); ReadLn(NrF); { la fel pt. fete }
  for i:=1 to NrF do
    begin Write('. numele fetei nr. ',i,' :'); ReadLn(NumeF[i]) end;
  for i:=1 to NrB do
     begin
       Punctaj[i]:=0;
       for j:=1 to NrF do
         begin
          { se memoreaza situatia baiat nr. i <-> fata nr. j
            in matricea "Iubit" si se actualizeaza punctaj baiat i }
          Write('Il iubeste ',NumeF[j],' pe ',NumeB[i], ' ? [da/nu] >');
          ReadLn(raspuns);
          if raspuns='nu' then Iubit[i,j]:=0 else Iubit[i,j]:=1;
          Punctaj[i]:=Punctaj[i]+Iubit[i,j]
         end
     end;
  { se ordoneaza descrescator baietii dupa punctaj }
  for i:=1 to Pred(NrB) do
    for j:=Succ(i) to NrB do
      if Punctaj[j]>Punctaj[i] then
         begin
         { schimb baiatul i cu baiatul j }
           aux_i:=Punctaj[i];Punctaj[i]:=Punctaj[j];Punctaj[j]:=aux_i;
           aux_s:=NumeB[i]; NumeB[i]:=NumeB[j]; NumeB[j]:=aux_s;
           for fata:=1 to NrF do
              begin
                aux_i:=Iubit[i,fata];
                Iubit[i,fata]:=Iubit[j,fata]; Iubit[j,fata]:=aux_i
```

```
                end
            end;
    WriteLn; WriteLn('Situatia generala este:'); { afisarea rezultatului }
    if Punctaj[1]=Punctaj[2] then
       begin
         WriteLn('Nu exista un cel mai fericit unic !');
         for i:=1 to NrB do
             WriteLn(NumeB[i]+' - iubit de ',Punctaj[i],' fete.')
       end
    else
       begin
         WriteLn('Fericitul este '+NumeB[1]+
                 '. Pe el il iubesc ',Punctaj[1],' fete.');
         if NrB>=2 then
            begin
              WriteLn('Ceilalti sunt:');
              for i:=2 to NrB do
                  WriteLn(NumeB[i]+'-iubit de ',Punctaj[i],' fete.')
            end
       end; ReadLn
end.
```

Exerciții cu tablouri, date la concursuri

Prezentăm enunțurile (eventual puțin modificate) ale unor probleme date la Olimpiada de Informatică, faza județeană, în anul 1994, în a căror rezolvare sunt folosite cunoştințele învătate până acum.

1. (Alba) Un elev din clasa I are la dispoziție `n` litere mici din alfabetul latin, din care `m` distincte. Doamna învățătoare îi cere următoarele lucruri:
a) Să verifice dacă există litere care apar de mai multe ori şi să păstreze toate literele distincte o singură dată.
b) Să aşeze aceste litere în ordine alfabetică.
Exemplu: `n`=6, a,b,a,d,c,c sunt cele 6 litere. a) litere distincte: a,b,d,c; b) ordinea alfabetică: a,b,c,d.
2. (Botoşani) Pentru un grup de `n` persoane, se definesc perechi de forma (`i,j`) cu semnificația *persoana* `i` *este influențată de persoana* `j`. Se cere să se determine persoanele din grup care au cea mai mare influență.
3. (Braşov) O matrice cu `n` linii şi `n` coloane (`n` impar) care conține toate numerele naturale de la `1` la n^2 şi în care suma elementelor de pe fiecare linie, coloană şi diagonală este aceeaşi, se numeşte *pătrat magic* de ordinul *n*. De exemplu, un pătrat magic de ordinul 3 este:

6	1	8
7	5	3
2	9	4

Să se scrie un program care verifică - printr-o singură parcurgere - dacă o matrice cu `n` linii şi `n` coloane este pătrat magic.
4. (Braşov) Se dă un şir *a* cu `n` elemente numere întregi. Să se scrie un program care determină ce element se află pe poziția `k` (dată) în şirul obținut din şirul `a` prin ordonare, fără a-l ordona pe `a` şi fără a utiliza alte şiruri.
5. (Călăraşi) Numerele C_1 , C_2 , C_3 , ..., C_n reprezintă valorile colesterolului a `n` persoane. O persoană este considerată sănătoasă dacă are colesterolul între două limite date `a` şi `b`. Se cere să se afle numărul persoanelor sănătoase şi media aritmetică a colesterolului persoanelor bolnave.
6. (Dolj) O persoană are de cumpărat din `m` magazine `n` produse care au prețuri diferite. Să se întocmească un program care să indice pentru fiecare produs magazinul în care acesta are prețul minim. Cunoscând cantitățile ce trebuie cumpărate din fiecare produs, să se determine suma ce urmează a fi cheltuită.
7. (Dolj) Se consideră `n` aruncări cu un zar. Se cere:
a) Să se determine de câte ori apare fiecare față şi la a câta aruncare.

b) Să se determine toate perechile de forma (F_i, K_i), cu proprietatea că F_i+K_i este un număr par, unde F_i reprezintă numărul feței, iar K_i = numărul de apariții ale feței F_i .
8. (Mureş) Se dă un număr natural `N`. Să se verifice dacă numărul obținut prin eliminarea primei şi ultimei cifre din `N` este pătrat perfect.
10. (Sălaj) Să se determine cel mai mare divizor comun a `n` numere.

7. Tipul înregistrare

Necesitate
În primul capitol, când ordonam elevii, am spus că nu ne interesează decât înălțimile, nu şi numele lor. Dacă însă, dorim să facem un program în care numele elevilor şi eventual alte informații despre ei să fie luate în considerare, putem să utilizăm mai mulți vectori, de diferite tipuri pentru componente. Mai simplu ar fi, însă, utilizarea unui vector de *elevi*, unde, prin `elev` să înțelegem un nou tip de dată, nestandard, pe care să-l definim noi. Pentru aceasta, să vedem, din toate datele ce caracterizează un elev (toate *atributele* sale), care sunt cele mai importante pentru dezvoltarea unei aplicații de genul celei pentru un concurs de admitere în liceu: *numele, prenumele, inițiala tatălui, sexul, nota la limba română, nota la matematică, media* şi faptul dacă a fost *admis* sau nu. De aceea, vom defini un elev ca o **înregistrare** (**articol**), adică o "cutiuță", care are mai multe **câmpuri** ce pot fi de diferite tipuri:

```
type elev = record
              nume, prenume: String; init_tata, sex: Char;
              nota1, nota2, media: Real; admis: Boolean
            end;
```

Observăm că un astfel de tip grupează informații ce pot fi foarte diferite calitativ între ele, aşa încât, spre deosebire de vectori, ne va trebui să o altă modalitate de a referi un anumit câmp.

Definire. Referirea unui element
În general, un **tip înregistrare** se defineşte astfel:

```
type id_tip_inreg = record
                      lista_de_câmpuri_1: id_tip_1;
                      lista_de_câmpuri_2: id_tip_2;
                      ..........................
                      lista_de_câmpuri_n: id_tip_n
                    end;
```

Putem declara o variabilă de tip înregistrare prin `var X: id_tip_inreg`, dar puteam scrie de la bun început: `var X: record ... end`, păstrând anonimatul acestui tip. Dacă `c` este un câmp al unei variabile de tip înregistrare `X`, atunci referirea la `c` se poate face prin `X.c`.
Asupra variabilelor înregistrare se pot face doar operații de atribuire. Putem însă lucra cu diferitele câmpuri ale acestora, asupra cărora se pot executa toate operațiile acceptate de datele de respectivul tip.

Exemple
- Am văzut mai sus ce este un *elev*. O *clasă* este un vector de elevi, deci putem scrie ceva de genul : `type clasa = array[1..36] of elev`. Dacă avem o clasă `var C: clasa` şi un elev oarecare `var E: elev`, putem scrie ceva de genul: `C[3]:=E`, care înseamnă că cel de al treilea elev din clasa `C` va fi `E`. Cum ne referim la numele celui de al V-lea elev din această clasă? Prin `C[5].nume`. Putem face, de pildă, o atribuire de genul: `C[5].nume:='Mihalache'`.
- Să observăm că o clasă are totuşi un şef, care este un elev. De asemenea, ea are şi un diriginte, despre care ne interesează doar cum îl cheamă. Astfel, putem să reconsiderăm ideea de clasă şi să definim acest tip de date ca o înregistrare. Această înregistrare va avea patru câmpuri: un şir de caractere (numele dirigintelui), un elev (separat, şeful clasei), precum şi

întregul şir al celorlalți elevi, care va avea lungimea memorată într-un câmp de tip întreg (numărul elevilor din clasă).

```
type clasa = record
               diriginte: String;
               sef: elev;
               n: Integer;
               EL: array[1..36] of elev
             end
```

- Fireşte, o şcoală poate fi considerată ca fiind o mulțime de clase:

```
type scoala = array[1..20] of clasa;
```

sau, dacă ne interesează şi numele şcolii sau al directorului:

```
type scoala = record
                nume, director: String;
                C: array[1..20] of clasa
              end;
```

Fie `var S: scoala`. În ipoteza că am definit clasa şi şcoala ca înregistrări, cum ne-am putea referi la prenumele elevului al VII-lea din clasa a XV-a din şcoala `S`? Avem, succesiv: `S` este scoala. `S.C` este mulțimea claselor sale. `S.C[15]` este cea de a cincisprezecea clasă din şcoala `S`. Elevii acestei clase sunt reprezentați de `S.C[15].EL`. Fireşte, cel de al şaptelea elev din clasă va fi `S.C[15].EL[7]`, iar prenumele său va fi `S.C[15].EL[7].prenume`.

- Să presupunem că avem un nou tip de dată, pentru a defini un salariat dintr-o întreprindere:

```
type salariat =  record nume_si_prenume: String; salariu: Integer end;
```

O secție cuprinde 20 salariați:

```
type sectie = array[1..20] of salariat;
```

iar o întreprindere are 5 secții:

```
type intreprinere = array[1..5] of sectie.
```

Fie o întreprindere `var Intrep: intreprindere`. Pe cel de al treilea salariat din secția 2 îl cheamă *Popescu*. Cum precizăm acest lucru? Prin `Intrep[2][3]:='Popescu'`. Afişarea salariului lui Popescu se face prin `WriteLn(Intrep[2][3].salariu)`.

Instrucțiunea WITH

Să presupunem că avem de făcut un program pentru un concurs de admitere, unde pentru fiecare elev trebuie să memorăm numele ('ncluzând şi prenumele şi inițiala tatălui), cele două note, precum şi media finală de concurs. Afişarea rezultatului elevului al `i`-lea, în ipoteza că am lucrat cu un vector `elevul: array[1..20] of elev` s-ar face prin:

```
WriteLn(elevul[i].nume,elevul[i].nota1:6:2,
        elevul[i],nota2:6:2,elevul[i].media:6:2).
```

Totul este OK, ceea ce ne deranjează este faptul că trebuie să scriem de mai multe ori numele variabilei înregistrare: `elevul[i]`, pentru fiecare câmp în parte. Acest lucru se poate simplifica folosind instrucțiunea `with`, în felul următor:

```
with elevul[i] do
     WriteLn(nume, nota1:6:2, nota2:6:2, media:6:2);
```

Formatul general al instrucțiunii **`with`** este:

```
with câmp do  instr
```

`câmp` este un câmp al variabilei înregistrare în cauză, iar `instr` este o instrucțiune oarecare, deci poate fi şi o instrucțiune compusă, care să cuprindă, între `begin` şi `end` mai multe instrucțiuni, unele folosind respectiva variabilă înregistrare, iar altele nu.

Putem folosi instrucțiunea `with` şi pentru mai multe câmpuri, chiar de la înregistrări diferite, însă numai dacă nu există riscul unor confuzii.

Program demonstrativ

În continuare prezentăm o nouă variantă a programului pentru concursul de admitere în liceu, care foloseşte, de această dată, un vector de înregistrări. Fireşte, programul nu poate fi considerat decât din punct de vedere didactic, deoarece el lucrează cu cel mult 20 de elevi. Acest număr poate fi mărit, însă nu prea mult, căci se va afişa o eroare de depățire a memoriei.

Apoi mai există şi alte inconveniente, legate, în primul rând, de faptul că informațiile nu pot proveni şi nu pot fi stocate pe un suport extern. În lecțiile următoare, odată cu prezentarea unor noi tipuri de date (liste şi fişiere), aceste inconveniente vor fi eliminate.

Referitor la acest program, putem observa apariția a două variabile de control a execuției programului: una logică, care spune dacă datele (unei serii de elevi) au fost introduse sau nu, precum şi una întreagă (`optiune`), prin care se precizează ce operație se va executa. De asemenea, această a doua variabilă este folosită atunci când, deşi datele despre elevi sunt introduse, dorim să le reintroducem.

Când se realizează o ordonare a elevilor (indiferent de *cheie* (criteriu), are loc o interschimbare între elevi. Acest lucru se realizează cu ajutorul unei variabile auxiliare, care, fireşte va fi de acelaşi tip, deci de tip `elev`.

Propunem cititorului să înfrumusețeze acest program, aşa încât să permită listări şi după alte criterii (cele două note, elevii cu medii peste o anumită valoare etc.), precum şi un meniu ceva mai atractiv. Succes!

```
program AdmitereLiceu; { varianta cu "record" }
uses Crt;
type elev = record
              nume: String;
              nota1,nota2,media: Real
            end;
var elevul: array[1..20] of elev;
    numele: String; { numele elevului curent, completat cu spatii }
    i,j,NrElevi: Integer;
    aux: elev; optiune: Integer; date_introduse: Boolean;
{interschimbarea intre doi elevi se va face folosind aceasta variabila}
begin
  date_introduse:=False;
  repeat
    ClrScr;
    WriteLn('ORDONARE ELEVI LA CONCURS'); WriteLn;
    WriteLn('0 = oprire program');
    WriteLn('1 = introducere date elevi');
    WriteLn('2 = listare dupa medii');
    WriteLn('3 = listare alfabetica');
    WriteLn; Write('Optiunea dvs. = '); ReadLn(optiune);
    case optiune of
      1: begin
          if not date_introduse then optiune:=1
          else
            begin
              WriteLn('Atentie !  Aveti datele deja introduse !');
              WriteLn('Doriti reintroducerea lor ? (1=da, 2=nu)');
              ReadLn(optiune);
            end;
          if optiune=1 then
             begin
               Write('Dati numarul de elevi : '); ReadLn(NrElevi);
               for i:=1 to NrElevi do
                   with elevul[i] do
                     begin
                       WriteLn('Dati numele+prenumele!'); ReadLn(nume);
                       WriteLn('Dati notele !'); ReadLn(nota1, nota2);
                       media:=(nota1+nota2)/2
                     end;
               date_introduse:=True
             end
         end;
      2: if not date_introduse then
            WriteLn('Introduceti, mai intai datele despre elevi !')
         else
           begin
             for i:=1 to Pred(NrElevi) do
```

```
                for j:=Succ(i) to NrElevi do {comparatie dupa medii}
                    if elevul[i].media < elevul[j].media then
                       { are loc interschimbarea elevilor }
                       begin
                         aux:=elevul[i]; elevul[i]:=elevul[j];
                         elevul[j]:=aux
                       end;
              for i:=1 to NrElevi do
                  with elevul[i] do
                       begin
                         {se completeaza numele sau cu spatii la dreapta}
                         numele:=nume;
                         for j:=1 to 40-Length(nume) do
                             numele:=numele+' ';
                         WriteLn(i:2,'. ',numele,' : ', nota1:6:2,
                                 nota2:6:2, media:6:2)
                       end
            end;
        3: if not date_introduse then
              WriteLn('Introduceti, mai intai datele despre elevi !')
           else
             begin
               for i:=1 to Pred(NrElevi) do
                 for j:=Succ(i) to NrElevi do {comparatie dupa nume}
                     if elevul[i].nume > elevul[j].nume then
                        { are loc interschimbarea elevilor }
                        begin
                          aux:=elevul[i]; elevul[i]:=elevul[j];
                          elevul[j]:=aux
                        end;
               for i:=1 to NrElevi do
                   with elevul[i] do
                        begin
                         { se completeaza numele sau cu spatii la dreapta }
                          numele:=nume;
                          for j:=1 to 40-Length(nume) do
                              numele:=numele+' ';
                          WriteLn(i:2,'. ',numele,' : ',
                                  nota1:6:2, nota2:6:2, media:6:2)
                        end
          end;
      end;
      WriteLn; WriteLn('Apasati <Enter> pentru continuare !');
      ReadLn
    until optiune=0
  end.
```

Exerciții

1. Scrieți declarații de tip pentru a specifica:

a) o dată calendaristică;

b) timpul în ore, minute şi secunde;

c) un număr complex (vezi *Algebra, cl. a X-a*!);

d) un punct în planul euclidian, dat prin coordonatele sale.

2. Să se scrie un program care să simuleze amestecarea unor cărți de joc între "`9`" şi "`As`".

În continuare prezentăm două exerciții împreună cu rezolvările lor, punînd în evidență unele aspecte teoretice.

3. Să se scrie un program care mişcă simultan mai multe bile pe ecran.

Rezolvare. După cum observați, revenim asupra acestei probleme, de data aceasta putând memora coordonatele unei bile într-o înregistrare, acest tip de date foarte flexibil. Vom folosi, aşadar, un vector de bile! Ceea ce mai trebuie remarcat, în programul următor, este utilizarea instrucțiunii `with`, care în acest caz ne simplifică mult munca. De asemenea, pentru a fi

sugestivi, am definit tipul `TBila`, iar variabila vector am numit-o `Bila`, deci am folosit litera "`T`" pentru *tip*. Acest lucru vi-l recomandăm şi dumneavoastră, mai ales când lucrați cu date de tip *înregistrare*.

```
program MaiMulteBile_varianta2;
uses Crt;
const max=20;
type TBila = record X,Y: Integer; Culoare: Byte; dX,dY: -1..1 end;
var Bila: array[1..max] of TBila; n,i: Integer;
begin
  ClrScr; Write('Numarul de bile = '); ReadLn(n); ClrScr;
  { initializare atribute bile }
  for i:=1 to n do
      with Bila[i] do
           begin
             X:=Random(77)+2; Y:=Random(22)+2; Culoare:=1+Random(15);
             dX:=Random(2); if dX=0 then dX:=-1;
             dY:=Random(2); if dY=0 then dY:=-1
           end;
  repeat
    for i:=1 to n do
        with Bila[i] do
             begin
             { sterge bila i din pozitia veche si afiseaz-o
               in cea noua, ai grija la margini! }
               GoToXY(X,Y); Write(' '); X:=X+dX; Y:=Y+dY;
               if (X=1) or (X=80) then
                  begin Sound(500); Delay(10); NoSound; dX:=-dX end;
               if (Y=1) or (Y=25) then
                  begin Sound(100*i);Delay(10); NoSound; dY:=-dY end;
               TextColor(Culoare); GoToXY(X,Y); Write(#15);
             end;
    Delay(50)
  until KeyPressed;
  ClrScr
end.
```

4. Să se realizeze un program care să deseneze o fereastră (în sens informatic!) pe ecran. Fereastra va fi specificată prin colțurile stânga-sus şi dreapta-jos, prin numele său şi prin culorile de desenare.

Rezolvare. Tipul înregistrare ne permite declararea unui tip `TFereastră`, pentru obiecte dreptunghiulare, cu chenar şi titlu în partea de sus:

```
type TFereastra = record
                    x1,y1,x2,y2: Integer;
                    cul1, cul2: Byte;
                    nume: String
                  end.
```

Următorul program desenează o fereastră pe ecran. El foloseşte o procedură din biblioteca `Crt`: **`Window`**. Cu cei patru parametri ea stabileşte o zonă de ecran care se comportă ca ecranul însuşi, deci orice afişare va fi relativă la colțul stânga-sus al ferestrei definite. Revenirea la coordonate absolute se face prin `Window(1,1,80,25)`. Noi vom folosi această procedură doar pentru a şterge mai rapid porțiunea de ecran respectivă. Scrierile le vom face în coordonate absolute. Cititorul poate exersa procedura singur, dacă o găseşte utilă. De pildă, pentru a controla scrierea unui text în interiorul ferestrei, se va folosi mai întâi un apel al lui `Window`, aşa cum am procedat şi noi în programul de mai jos.

```
program Fereastra1;
uses Crt;
{ mai jos sunt definite constante pentru caractere grafice
  ce permit desenarea unui chenar dublu }
const vert=#186; dr_sus=#187; dr_jos=#188;
      st_jos=#200; st_sus=#201; oriz=#205;
type TFereastra = record
```

```
                    x1,y1,x2,y2: Integer;
                    cul1, cul2: Byte; nume: String
                  end;
var Fereastra: TFereastra; x,y: Integer;
begin
  ClrScr;
  with Fereastra do
       begin
        { initializare câmpuri }
        x1:=20; x2:=70; y1:=5; y2:=20; cul1:=Yellow; cul2:=Blue;
        nume:='Test de fereastra';
        { - stergerea ferestrei in culoarea de fond cul2 }
        Window(x1,y1,x2,y2); TextBackground(cul2); ClrScr;
        Window(1,1,80,25); { - revenirea la coordonatele absolute }
        TextColor(cul1); { - selectarea culorii cul1
                           pentru desenarea chenarului }
        { - desenarea marginilor de sus si jos }
        for x:=x1+1 to x2-1 do begin
            GoToXY(x,y1); Write(oriz); GoToXY(x,y2); Write(oriz) end;
        { - desenarea marginilor din stanga si dreapta }
        for y:=y1+1 to y2-1 do begin
            GoToXY(x1,y); Write(vert); GoToXY(x2,y); Write(vert) end;
        { afisarea colturilor }
        GoToXY(x1,y1); Write(st_sus); GoToXY(x1,y2); Write(st_jos);
        GoToXY(x2,y1); Write(dr_sus); GoToXY(x2,y2); Write(dr_jos);
        { afisarea numelui ferestrei, pe centru,
          incadrat de doua spatii }
        GoToXY(x1+(x2-x1) div 2 - (1+Length(nume)) div 2, y1);
        Write(' '+nume+' '); Window(x1+1,y1+1,x2-1,y2-1);
        Write('   Acest text este scris doar in fereastra definita ');
        Write('de noi, el neputand sa depaseasca limitele '+
              'acesteia ...');
    end;
  Window(1,1,80,25); ReadLn; TextColor(7); TextBackground(0); ClrScr
end.
```

8. Tipul mulțime

Necesitate

Să presupunem că vrem să scriem un program care să se termine la apăsarea tastei 's', fie ea literă mică sau mare, sau la apăsarea tastei 'q', literă mică sau mare. Putem scrie ceva de genul:

```
repeat
     tasta:=ReadKey;
     ..................
until (tasta='s') or (tasta='S') or (tasta='q') or (tasta='Q');
```

Această condiție de terminare se poate scrie mai simplu: `tasta in ['s','S','q','Q']`. Acest lucru se traduce prin: *tasta aparține mulțimii conținând caracterele* 's','S', 'q' *și* 'Q'.

Există multe situații în care definirea unui tip mulțime fie că este foarte necesară, fie că ne simplifică munca de programator.

Definire. Operații. Exemple

Un tip mulțime se defineşte în raport cu un tip de bază care trebuie să fie un tip ordinal (scalar). Fiind dat un asemenea tip de bază, tipul mulțime reprezintă mul]imea tuturor submulțimilor tipului de bază, inclusiv mulțimea vidă. Fiecare valoare a tipului mulțime este o mulțime ai cărei membri (în număr maxim de 255) sunt valori distincte ale tipului de bază.

- Un **tip mulțime** se defineşte astfel:

```
type id_tip_multime = set of id_tip;
```

unde `id_tip_multime` este identificatorul tipului mulțime care se defineşte, iar `id_tip` identifică un tip ordinal (deci poate fi `Integer`, `Char`, `Boolean`, nu şi `Real` sau `String` sau

vreun tip structurat (`array` sau `record`). Astfel dacă declarăm o variabilă `var A: id_tip_multime`, atunci `A` va fi o mulțime de elemente de tipul `id_tip`.

- *Exemple*:

Putem defini mulțimi pentru cifrele din baza zece:

```
type MultCifre = set of Integer;
var Par, Impar: MultCifre
```

dar putem scrie direct: `var Par, Impar: set of Integer`.

- O mulțime poate fi specificată enumerându-i-se elementele între paranteze pătrate, care țin locul acoladelor din matematică.

Astfel, putem considera: `Par:=[0,2,4,6,8]; Impar:=[1,3,5,7,9]`.

Putem folosi şi subdomenii: de exemplu: `var Litera, Cifra: set of Char;`

`Litera := ['a'..'z', 'A'..'Z']; Cifra := ['0','1','2'..'8','9'].`

- Mulțimea vidă, indiferent de ce tip, va fi notată prin `[]`.
- Operațiile ce se pot face cu mulțimi (de acelaşi tip) sunt *reuniunea*, notată prin "+", *intersecția*, notată prin "*", precum şi *diferența*, notată prin "**-**".

Să dăm câteva *exemple*:

1. Reuniunea

```
program Multime1;
var Par, Impar, Cifra: set of Integer;
begin
  Par := [0,2,4,6,8]; Impar := [1,3,5,7,9]; Cifra := Par + Impar
end.
```

2. Intersecția

```
program Multime1;
var Vocala, Cuvant, Inters: set of Char;
begin
  Vocala := ['a','e','i','o','u','y'];
  Cuvant := ['l','i','t','o','r','a','l'];
  Inters := Vocala * Cuvant
end.
```

3. Diferența

```
program Multime2;
var Vocala, Cuvant, Dif1, Dif2, DifSimetrica: set of Char;
begin
  Vocala := ['a','e','i','o','u','y'];
  Cuvant := ['g','e','r','o','v','i','t','a','l'];
  Dif1 := Vocala - Cuvant; Dif2 := Cuvant - Vocala;
  DifSimetrica := Dif1 + Dif2
end.
```

- De asemenea, avem relații de *egalitate* ("="), *inegalitate* ("<>"), *incluziunea* (notată prin "<=" şi ">="). Apartenența unui element la o mulțime se notează prin cuvântul rezervat "**in**".

4. Relațiile de incluziune

Avem: `['a','e','i'] <> ['a','e','o','u']` este `True`; `[1,3,6] = [6,3,1]` este `True`. `[2,4,6] <= [1..6]` este `True`, iar `[3,5]>=[1,2]` este `False`.

5. Apartenența

```
program TestLitera;
var Vocala: set of Char; lit: Char;
begin
  repeat
  Write('Dati litera : '); ReadLn(lit);
  if lit in Vocala then WriteLn('Este vocala !')
  else WriteLn('Nu este vocala ...')
  until lit in ['.','/']
end.
```

- Pentru a afişa elementele unei mulțimi `M` nu putem scrie direct `WriteLn(M)`. De asemenea, nu putem să-l citim pe `M` prin `ReadLn(M)`.

Să presupunem că `M` este o mulțime de caractere. Pentru a realiza citirea şi scrierea, vom proceda în felul următor: Vom pleca cu mulțimea `M` vidă şi îi vom adăuga câte un element citit. La afişare vom parcurge întregul tip `Char` şi, ori de câte ori întâlnim un caracter care se află în mulțime, (pe care o copiem în `Aux`), îl afişăm şi îl înlăturăm din mulțime. În final afişăm caracterul având codul `ASCII 8`, apoi un punct, astfel încât ultimul element va avea un punct şi nu o virgulă după el, cum se întâmpla în programele de până acum. Caracterul `#8` corespunde trecerii cursorului spre stânga cu o poziție.

```
program CitireAfisareMultimi;
var M, Aux: set of Char; lit: Char;
begin
  { citirea } M:=[];
  repeat
    Write('Dati un element al multimii (. si / = stop) -> ');
    ReadLn(lit); if not (lit in ['.','/']) then M:=M+[lit]
  until lit in ['.','/'];
  WriteLn('Multimea este:'); Aux:=M; lit:=Chr(0);
  while (lit <= Chr(255)) and (Aux <> []) do
    begin
      if lit in Aux then begin Write(lit,', '); Aux:=Aux-[lit] end;
      lit := Succ(lit)
    end;
  WriteLn(#8#8,'.'); ReadLn
end.
```

Program demonstrativ

Vom genera toate numerele prime cuprinse în domeniul de valori `2..N`, cu `N≥2`. Vom folosi algoritmul cunoscut sub numele de **ciurul** (**sita**) **lui Eratostene**. Vom pune în *sită* toate numerele cuprinse în domeniul `2..N`. Vom alege şi extrage din sită cel mai mic număr (care întotdeauna va fi prim), în mod repetat, cât timp sita nu este vidă. Acesta se include în mulțimea numerelor prime `prim`. Se elimină din sită toți multiplii acestui număr. Procesul se încheie când sita s-a golit.

```
program Eratostene1;
var N: Integer; elem,j: Integer; sita, prim: set of 2..50;
begin
  Write('Dati N = '); ReadLn(N);
  sita := [2..N]; prim := []; elem := 2;
  repeat
    while not (elem in sita) do elem := succ(elem);
    prim:=prim+[elem]; j:=elem; WriteLn(elem);
    while j<=N do { eliminare } begin sita:=sita-[j]; j:=j+elem end
  until sita=[]; ReadLn
end.
```

Ca variantă, putem considera mulțimile ca având doar numere impare şi numărul 2:

```
program Eratostene2;
var N,NN: Integer; sita, prim: set of 2..50; next,j,elem: Integer;
begin
  Write('Dati N = '); ReadLn(N); NN:=2*N;
  WriteLn('Numere prime pana la ',NN,':');
  sita := [2..N]; prim := []; WriteLn(2); next := 2;
  repeat
    while not (next in sita) do next := succ(next);
    prim:=prim+[next]; elem:=2*next-1;
    WriteLn(elem); j:=next;
    while j<=N  do { eliminare } begin sita:=sita-[j]; j:=j+elem end
  until sita=[]; ReadLn
end.
```

Exerciții

1. Se citesc `n` mulțimi de numere naturale mai mici sau egale cu 255. Se cere să se precizeze dacă aceste mulțimi sunt disjuncte sau nu două câte două. (Două mulțimi sunt *disjuncte* dacă nu au elemente comune).
2. Să se verifice dacă numele unei persoane este corect introdus (numele este un şir de caractere ce nu conține cifre).
3. Să se genereze mulțimile `A={x; x≤n , 7x+1 se divide prin 5}` **şi** `B={x; x≤n, x²+1 se divide prin 3}`, unde numărul întreg `n` se citeşte de la tastatură. Să se afişeze elementele intersecției lui `A` cu `B`.

9. Exerciții recapitulative

1. Să se scrie un program care să administreze un parc de automobile. Informațiile relative la un automobil sunt: numărul de locuri, puterea (în cai putere), marca, numărul de înmatriculare, tipul de carburant (benzină sau motorină), natura (berlină, break sau coupe). Programul să permită intrarea unei maşini, ieşirea unei maşini, înlocuirea unei maşini cu alta de acelaşi model (având alt număr de înmatriculare).
2. Se dă o propoziție în care cuvintele sunt separate fie prin spațiu, fie prin caracterele punct, virgulă, punct şi virgulă, semnul exclamării şi semnul întrebării. Despărțiți propoziția în cuvinte.
3. Dintr-o matrice dată `M`, să se listeze valorile tuturor punctelor *şa* şi pozi]ia lor. `M[i,j]` este considerat punct *şa* dacă este minim pe linia `i` şi maxim pe coloana `j`.
4. Scrieți un program care să citească temperaturile măsurate din oră în oră, precum şi cantitățile zilnice de precipitații dintr-o lună a anului. Apoi programul să permită afişarea temperaturii maxime (împreună cu ziua şi ora asociată), temperatura minimă (cu ziua şi ora asociată), lista zilelor, ordonată descrescător în funcție de cantitatea de precipitații, media precipitațiilor zilnice din respectiva lună a anului. Eventual să se realizeze histograme la diferite date statistice.
5. Să se elimine dintr-o matrice `A` (cu `m` linii şi `n` coloane) linia `l` şi coloana `k` şi să se listeze matricea rămasă. (Nu se va folosi o altă matrice.).
6. Să se bordeze o matrice `A` (având `m` linii şi `n` coloane) cu linia `m+1` şi coloana `n+1`, unde `A[m+1,j]` să fie suma elementelor de pe coloana `j`, cu `j` de la `1` la `n`, iar `A[i,n+1]` să fie suma elementelor de pe linia `i`, cu `i` de la `1` la `m`.
7. Fiind dat un vector `V` cu `n` componente elemente întregi, să se formeze un nou vector `W`, în care `W[i]` să conțină suma (produsul) cifrelor elementelor lui `V[i]`.
8. Să se definească un tip subdomeniu pentru literele mici ale alfabetului latin. Fie un vector `V` cu elemente de acest tip. Să se formeze un vector `W` care să conțină numai elementele din `V` ce sunt consoane şi să se afişeze toate majusculele corespunzătoare literelor din `W`.
9. Să se genereze aleator cuvinte de lungime 4. Se va folosi funcția `Chr`.
10. Să se insereze între oricare două elemente dintr-un vector de numere reale media lor aritmetică.
11. Să se determine de câte ori se întâmplă ca într-un vector de numere reale să existe un element ce reprezintă produsul elementelor sale vecine.
12. Să se scrie în spirală numerele de la `1` la n^2 într-o matrice pătratică cu `n` linii şi `n` coloane începând: a) din centrul matricei (`n` impar); b) din colțul stânga-sus. Se vor considera, pe rând, ambele sensuri de deplasare.
13. Se dă o matrice pătratică de ordin `n`. Se consideră că cele două diagonale împart matricea în patru zone: *nord*, *sud*, *est* şi *vest* (elementele de pe diagonală nu fac parte din nici o zonă).
a) Să se calculeze suma elementelor din *nord*, produsul elementelor din *sud*, media aritmetică a elementelor din *est* şi numărul elementelor nule din *vest*.
b) Să se obțină simetrica matricei inițiale față de diagonala principală.
c) Să se obțină simetrica matricei inițiale față de diagonala secundară.
d) Să se obțină simetrica matricei inițiale față de o axă orizontală ce trece prin centrul matricei.
e) Să se obțină simetrica matricei inițiale față de o axă verticală ce trece prin centrul matricei.

14. Considerând tipul definit prin enumerarea (*rosu, alb, maro, gri*) să se stabilească ce valori reprezintă:

```
a) Ord('Z')-ord(maro); b) Pred(Ord(gri));
c) Succ(rosu); d) Succ(Chr(Ord('B')+3-Ord(alb))).
```

Este corectă o instrucțiune de genul:

```
for c:=rosu to maro do WriteLn(c), unde c: (rosu, alb, maro, gri) ?
```

Dar `for c:=alb to maro do WriteLn(Ord(c))` ?

15. Determinați erorile de sintaxă din următorul program, fără a utiliza calculatorul:

```
program Test Erori;
type vector = array[1.....n] of Integer;
var i: Integer; x: vector;
begin
       for i:=1 to n do ReadLn(x[i]);
       for i:=1 to n do
            x[i]:=x[i]/2;
       endfor
end.
```

16. Pentru o matrice cu `m` linii şi `n` coloane, cu elemente numere reale, să se listeze toate liniile, precum şi numerele sale de ordine, care conțin cel puțin `k` elemente nule.

17. Scrieți un program care verifică dacă o matrice este simetrică față de diagonala principală sau dacă are toate elementele nule.

18. Se dau două şiruri de numere întregi `x` cu `nx` elemente şi `y` cu `ny` elemente, unde `nx>ny`. Să se decidă dacă `y` este un subşir al lui `x`, adică dacă un număr `k`, astfel încât:
$x_k = y_1$, $x_{k+1} = y_2$, ..., $x_{k+ny-1} = y_{ny}$. În caz afirmativ, se va afişa valoarea lui `k`.

19. Pentru două matrice cu elemente reale, `A`, cu `m` linii şi `n` coloane, şi `B`, cu `n` linii şi `p` coloane, se numeşte *produsul matricelor* `A` *şi* `B` matricea `C`, cu `m` linii şi `p` coloane, în care elementul `C[i,j]` este suma `A[i,1]×B[1,j]+A[i,2]×B[2,j]+...+A[i,n]×B[n,j]`, pentru orice `i` între `1` şi `m` şi orice `j` între `1` şi `p`. Să se scrie un program care citeşte două matrice `A` şi `B` şi calculează şi afişează produsul lor, `C`.

20. Fie dat un vector $x=(x_1, x_2, ..., x_n)$. Să se modifice vectorul astfel încât, în final, să avem:

a) $x=(x_2, x_3, ..., x_n, x_1)$;
b) $x=(x_n, x_1, ..., x_{n-1})$;
c) $x=(x_2, x_1, x_4, x_3, ..., x_n, x_{n-1})$ (n - par).

21. Să se transforme un număr din baza `10` în baza `2`, memorând numărul rezultat:
a) într-un vector; b) într-un şir de caractere.

22. Să se scrie un program pentru admiterea la un liceu care să permită:
a) inițializarea bazei de date (a vectorului de înregistrări de tip elev);
b) adăugarea unui elev în baza de date;
c) înlocuirea unui elev cu alt elev;
d) inserarea unui elev pe o anumită poziție în baza de date;
e) eliminarea unui elev de pe o anumită poziție din baza de date;
f) eliminarea din baza de date a unui elev având un anumit nume;
g) căutarea unui elev după nume;
h) calcularea mediilor;
i) listarea alfabetică a elevilor;
j) listarea elevilor în ordinea descrescătoare a mediilor;
k) listarea tuturor elevilor cu medii peste o valoare dată;
l) amestecarea elevilor din baza de date;
m) eliminarea ultimului elev din baza de date.

23. Să se realizeze deplasarea unei ferestre pe ecran, folosind tastele de cursor şi, de asemenea, redimensionarea sa, la apăsarea altor două taste (de pildă `Home` şi `End`).

24. Să se realizeze diferențele a două mulțimi, memorate sub forma a doi vectori, precum şi reuniunea diferențelor, numită *diferență simetrică*.

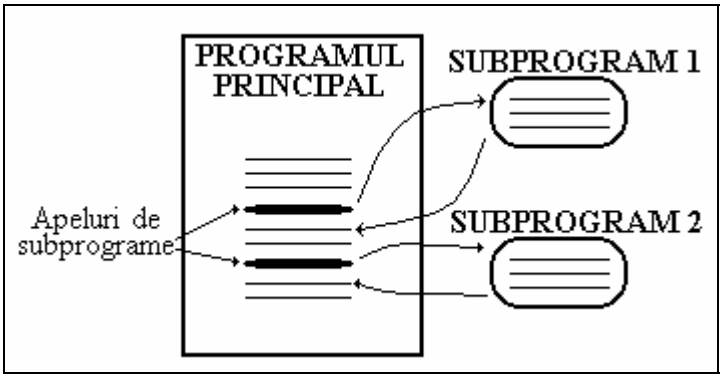
25. Se dau două polinoame sub forma a doi vectori. Să se determine suma, diferența, şi produsul lor.
26. Să se împartă două polinoame care sunt memorate sub forma a doi vectori, obținând încă doi vectori, care să păstreze câtul şi respectiv restul împărțirii.
27. Să se realizeze un joc de tip "*puzzle*" ("*perspico*"), în care jucătorul dispune de o matrice cu `4×4` căsuțe. În acestea sunt puse, pe rând, nişte pătrățele cu numerele 1, 2, ..., 15, ultima căsuță fiind liberă. Se amestecă aceste numere, prin deplasarea căsuței libere sus, jos, stânga sau dreapta. Simulați amestecarea pătrățelelor şi dați posibilitatea utilizatorului să refacă matricea, cu ajutorul tastelor de cursor. Generalizare pentru `n×n` căsuțe..
28. Să se definească tipul de date `complex` şi să se scrie un program care să efectueze calcule cu numere complexe. (vezi *Algebra*, cl. a X-a).
29. Numerele de la `1` la `n` sunt aşezate în ordine crescătoare pe circumferința unui cerc, astfel încât `n` ajunge lângă `1`. Începând cu numărul `s`, se elimină din cerc, numerele din `k` în `k`, după fiecare eliminare, cercul strângându-se. Care va fi numărul ce va rămâne?
30. Aceeaşi problemă ca la 29 dar numerele nu se elimină, ci se marchează, până unul din ele va fi marcat de două ori. Câte numere au rămas nemarcate?
31. Fie `A` o matrice cu `m` linii şi `n` coloane, cu elemente reale. Să se determine linia `l` şi coloana `k` pe care suma elementelor este maximă.
32. Fie `A` o matrice cu `m` linii şi `n` coloane, cu numere reale şi un vector `V` cu `n` componente reale. Să se determine dacă acest vector apare ca linie în matricea `A` şi, în caz afirmativ, să se afişeze numărul acestei linii.
33. Să se determine toate soluțiile de tip `Byte` ale ecuației `7x-4y=3`.
34. Fie `V` un vector cu elemente de tip `Char`. Să se construiască un vector `W`, astfel încât `W[i]` = numărul de apariții ale lui `V[i]` în vectorul `V`.
35. Se dă un vector de numere întregi pozitive. Să se determine cea mai lungă subsecvență de elemente consecutive prime, ale căror oglindite sunt tot numere prime.
36. Se dă un vector de numere întregi. Se cere să se afişeze subsecvența palindromică de lungime maximă. (Elementele subsecvenței sunt elemente consecutive în vector.)

Lecția 5
SUBPROGRAME

1. Funcții şi proceduri

De multe ori e necesar ca o anumită secvență de instrucțiuni să se execute de mai multe ori în cadrul unui program, eventual pentru alte date. De aceea e bine ca aceste instrucțiuni să fie grupate într-un modul purtând un nume. El va fi numit **subprogram**.

Procedurile despre care vorbeam în primul capitol nu erau altceva decât nişte subprograme. Astfel, ele sunt apelate din cadrul programului sau chiar din cadrul altui subprogram. Ambele se numesc **blocuri**. Singura condiție impusă este aceea ca *dacă un subprogram* `P` *apelează un subprogram* `Q`*, atunci se scrie întâi subprogramul* `Q` *şi apoi subprogramul* `P`.
Când se apelează un subprogram, executarea continuă cu prima instrucțiune din respectivul subprogram. Când se termină executarea instrucțiunilor din subprogram, se continuă cu instrucțiunea imediat următoare apelului, din blocul apelant, conform schmei de mai jos.



Nu există restricții în apelul subprogramelor. Un subprogram poate fi apelat de mai multe ori în cadrul aceluiaşi bloc, chiar de el însuşi, caz în care apelul este **recursiv**.
Există două tipuri de subprograme în limbajul Pascal: proceduri şi funcții.
Funcțiile sunt subprograme care returnează o valoare, adică, la sfârşitul executării lor, în blocul apelant se dispune de o anumită valoare, valoarea returnată de respectiva funcție. Astfel, dacă avem o funcție *F*, un apel de funcție se poate folosi ca orice expresie, având tipul valorii returnate de funcție, deci poate fi folosit oriunde se poate folosi o expresie de acel tip: de exemplu: `x:=F; WriteLn(F).` De fapt vom apela la funcțiile definite de noi aşa cum apelam, până acum, orice gen de funcție: `Sin`, `Trunc`, `Odd`, `Length`, `Pos` etc.. Procedurile sunt cazuri particulare de funcții, motiv pentru care prezentăm mai întâi funcțiile. Ele se comportă ca nişte funcții oarecare, dar nu returnează valori. Aceasta nu înseamnă că nu pot comunica cu blocul apelant. Apelul procedurilor se face ca a oricăror proceduri folosite deja în programele noastre de până acum: `WriteLn`, `ReadLn`, `Insert` etc..

Folosind opțiunea de compilare `{$X+}` (`Extended syntax`), putem folosi funcțiile ca şi proceduri, neinteresându-ne valorile returnate de ele, ci doar *ce fac* ele. Procedurile şi funcțiile se pot apela între ele. Mai mult, putem avea atât proceduri, cât şi funcții recursive.

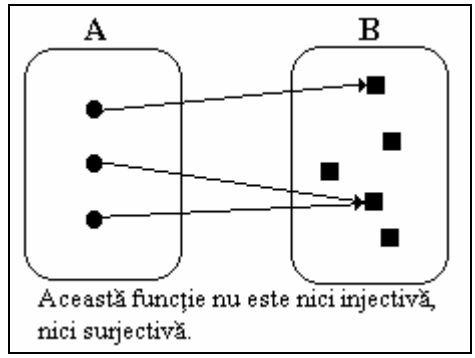
Subprogramele se definesc, în întregime, în partea declarativă a unui program, deci înainte de corpul propriu-zis al programului, astfel:

```
antet de subprogram;
declarații proprii subprogramului
begin
        instrucțiuni
end;
```

Antetul de subprogram diferă de la funcție la procedură.

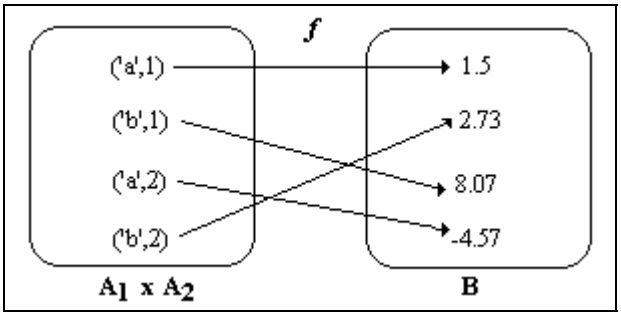
Funcții - o paralelă matematică - informatică

Vom reaminti noțiunea de funcție aşa cum apare ea în matematică şi vom prezenta diverse extinderi ale acesteia. Astfel, să considerăm două mulțimi *oarecare* (deci nu neapărat de numere !) `A` şi `B`. În matematica elementară se numea *funcție* o lege `f` (o corespondență) prin care fiecărui element din `A` i se asociază un *unic* element din `B`. Astfel, această definiție nu exclude posibilitatea ca să existe în `A` două elemente cărora să le corespundă un acelaşi element din `B`. Dacă nu există un asemenea caz, se spune că funcția este *injectivă*. De asemenea, în `B` pot rămâne elemente care să nu corespundă nici unui element din mulțimea `A`. Dacă nu s-ar întâmpla aşa ceva, funcția s-ar numi *surjectivă*. O funcție injectivă şi surjectivă se numeşte funcție *bijectivă* sau *bijecție*.



`A` poate conține orice gen de elemente. Poate fi mulțimea numerelor întregi sau reale (`Integer`, `Real`), sau chiar o mulțime de caractere, şiruri de caractere sau valori logice (`Char`, `String`, `Boolean`). Astfel, am putea avea o funcție de genul `f:Real→Integer`, ceea ce ar corespunde în matematică unei declarații $f:\mathbf{R}\rightarrow\mathbf{Z}$. Am putea avea şi funcții de genul `f:Char→Boolean`, fără corespondent în matematică. Această declarație ar corespunde unei funcții care are argumentul un caracter, iar rezultatul o valoare logică. (De pildă, am putea avea o funcție care să verifice dacă un caracter este sau nu o vocală.).

Datele din `A` ar putea fi chiar şi tablouri sau înregistrări etc.. Nu acelaşi lucru este valabil şi pentru mulțimea `B`, ale cărei elemente nu pot fi de tip structurat. În plus, domeniul funcției (`A`), ar putea reprezenta produsul cartezian a două mulțimi, A_1, care să conțină elemente de un anumit tip t_1, şi A_2, cu elemente de un alt tip t_2. Astfel, un element din `A` ar fi o pereche ordonată de două elemente, primul din A_1, iar al doilea din A_2. De exemplu, fie A_1={'a','b'}, A_2={1,2}, iar `B`=**R**. Avem: `f((`'a',1`))=1.5`; `f((`'b',2`))=2.73`; `f((`'b',1`))=8.07`; `f((`'a',2`))=-4.57`.



Prezența celor două perechi de paranteze se datorează următoarelor:

- parantezele groase apar pentru a încadra argumentele funcțiilor;
- parantezele subțiri pentru a marca perechile ordonate ce sunt pe post de argumente.

În cele ce urmează vom renunța la una din perechile de paranteze, considerând că avem o funcție cu două argumente. E clar că putem să vorbim, în mod similar, de *funcții cu mai multe argumente, care pot fi chiar de tipuri diferite şi nu neapărat numerice*.
În matematică a calcula `f(2)` sau a calcula `f(x)`, unde `x` are valoarea `2` este acelaşi lucru. Se calculează `f(x)`, adică `f(2)`, iar `x` *nu-şi modifică valoarea*, în urma apelului funcției. La fel se întâmplă şi în informatică. Dar putem avea şi cazul în care *argumentele funcției să-şi modifice valorile lor inițiale*. Acest lucru nu ar putea fi posibil decât, firește, în cazul în care acele argumente ar fi doar *variabile*, nu şi constante sau expresii mai complicate. În informatică determinarea valorii unei funcții poate fi însoțită de *efecte laterale*, lucru care nici nu se pune în discuție în matematică. Putem avea, de asemenea, funcții fără argument, dar care returnează valori şi eventual, au unele efecte laterale. Valorile returnate pot depinde, în acest caz, de anumite teste asupra sistemului electronic de calcul (tastatură, memoria disponibilă, situații diferite ale discurilor).

Declararea funcțiilor. Parametri
O funcție numită `id_fun` se declară astfel:

```
function id_fun (v1,...vn: t1; w1,..., wm: t2; ...;
                 var z1,..,zp: t3; var u1, .., uq: t4; ...): tip_f;
declarații locale;
begin
        instrucțiune 1;
        instrucțiune 2;
        ......................
        instrucțiune k;
        id_fun := expresie
end;
```

În definiția de mai sus avem următoarele părți:

- antetul funcției:
  ```
  function id_fun (v1,...vn: t1; w1, ..., wm: t2; ...;
                   var z1,..,zp: t3; var u1, .., uq: t4; ...): tip_f;
  ```
- partea declarațiilor locale funcției: `declarații locale;`
- corpul funcției:
  ```
  begin
          instrucțiune 1;
          instrucțiune 2;
          ......................
          instrucțiune k;
          id_fun := expresie
  end;
  ```

Antetul funcției cuprinde, pe lângă numele funcției (`id_fun`), două grupe de **parametri formali** (**argumente**). Prima conține mai multe liste de parametri ale căror valori se pot modifica în contextul funcției, dar rămân nemodificate în afara ei. Valorile parametrii din listele precedate de cuvântul `var` se pot modifica, în sensul că după revenirea din apelul subprogramului, vom avea acces la noile valori (cele modificate în subprogram). **Parametrii efectivi** sunt acei parametri cu care se apelează funcția. Aceştia sunt puşi în corespondență directă cu parametri formali ai funcției, în ordinea în care sunt scrişi. Pentru listele din prima grupă, se pot transmite (ca parametri efectivi) orice gen de expresii de tipurile corespunzătoare, iar pentru parametrii de după *var* nu se pot transmite decât variabile. Nu există restricții cu privire la alegerea identificatorilor parametrilor formali, aşa încât putem folosi aceiaşi identificatori atât pentru parametrii formali, cât şi pentru cei efectivi (din blocul apelant). Prezența parametrilor nu este obligatorie.

Prin `tip_f` se înțelege **tipul rezultatului** returnat de funcție.

Partea declarațiilor locale funcției conține, ca şi blocul programului principal, orice declarație de tipuri, constante sau variabile. Trebuie însă să nu existe riscul unor confuzii. Astfel, nu putem redeclara aici o variabilă ce apare şi în lista parametrilor efectivi. Putem redeclara însă variabile care apar în blocul apelant cu acelaşi nume. În acest caz, în cadrul funcției, nu ne mai putem referi la variabilele respective din blocul apelant. De fapt există anumite reguli de vizibilitate a identificatorilor, asupra cărora vom reveni.

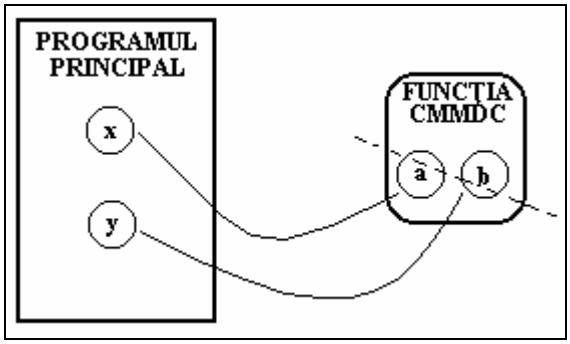
Corpul funcției cuprinde mai multe instrucțiuni. La funcții (nu şi la proceduri), trebuie ca să existe (la sfârşit) o instrucțiune de atribuire: `id_fun:=expresie`, unde `expresie` este de tipul `tip_f`. Aceasta pentru ca funcția să returneze o valoare.

Exemple

Pentru a clarifica noțiunile prezentate formal mai înainte, vom exemplifica.

1. Să scriem un program care să calculeze cel mai mare divizor comun a două numere. În acest scop, vom utiliza o funcție, care va avea două argumente: numerele întregi `a` şi `b`. În urma algoritmului, `a` şi `b` se modifică, însă acest lucru nu vrem să fie transmis înapoi în programul principal, aşa încât nu vom folosi cuvântul rezervat `var` în fața identificatorilor. În partea stângă a textului programului am numerotat instrucțiunile, în ordinea în care se execută ele.

```
                program CalculCMMDC;
A               function CMMDC(a,b: Integer): Integer;
                begin
3                       while a<>b do
                                if a>b then a:=a-b
                                else b:=b-a;
4                       CMMDC:=a
                end;
                var x,y: Integer;
                begin
1                       x:=20; y:=24;
2                       WriteLn(CMMDC(x,y))
                end.
```



În blocul principal avem două variabile globale `x` şi `y`, care se inițializează (în instrucțiunea `1`) cu valorile întregi `20`, respectiv `24`. În instrucțiunea `2`, avem un apel al procedurii `WriteLn`. Ea trebuie să afişeze o valoare întreagă, anume valoarea expresiei dintre paranteze. Aceasta este valoarea returnată de funcția `CMMDC`, care are parametrii efectivi `x` şi `y`, adică `20` şi `24`. De fapt, puteam scrie chiar şi `CMMDC(20,y)` sau `CMMDC(x,24)` sau `CMMDC(20,24)`. Se intră, în acest moment, în funcția `CMMDC`, unde, în punctul `A`, se creează două noi variabile*, cu numele* `a` şi `b`, care vor dispare la sfârşitul funcției. Ele sunt parametrii formali ai funcției. `a` va primi valoarea lui `x`, deci `20`, iar `b` pe cea a lui `y`, deci `24`. După încheierea ciclului `while` (`3`), `a` şi `b` vor avea ambele valoarea `4`. Aceasta este valoarea care este returnată de funcție (în instrucțiunea de atribuire `4`). Revenindu-se în blocul principal al programul, se va afişa, numărul `4`. Variabilele `x` şi `y` nu sunt afectate.

2. Următorul exemplu va face precizări asupra variabilelor care apar în blocul principal, parametrii funcției şi variabilele locale ei. Se calculează produsul `n! = 1×2×...×n`.

```
        program  Test1;
        function Factorial(n: Integer): Integer;
        var i,p: Integer;
        begin
                p:=1; for i:=1 to n do p:=p*i;
B               Factorial := p
        end;
        var p,i: Integer;
        begin
                p:=3;
C               i:=Factorial(p);
                WriteLn(i)
        end.
```

Programul va afişa `6` (`=3!`). Ce se întâmplă? În primul rând, avem declarată o funcție `Factorial`, având parametrul formal `n`. În interiorul funcției, se calculează produsul `p = n!`. Pentru aceasta se foloseşte încă variabilă locală `i`. La sfârşit, funcția returnează valoarea lui `p`, iar `i`, `p` şi `n` dispar.

Execuția programului se desfăşoară în felul următor: variabila globală `p` primeşte valoarea `3`. Se apelează funcția de calcul a factorialului, cu `p` global drept parametru efectiv. Astfel, parametrul formal `n` va primi valoarea `3`. Acest `p` global nu are nimic comun cu variabila locală funcției, purtând acelaşi nume. Deci, când execuția ajunge în punctul `B`, `p`-ul global va avea valoarea `3`, iar cel local valoarea `6`. În funcție apare o altă variabilă `i`, dar acest `i` nu este tot una cu variabila globală `i`, care primeşte valoarea factorialului lui `3`, deci `6`.

3. Să scriem, în continuare un program care să calculeze recursiv factorialul, adică ținând cont de formulele: `n!=1`, dacă `n=0` şi `n!=(n-1)! × n`, dacă `n>0`. Considerăm:

```
        function Fact(n: Integer): Integer;
        begin
                if n=0 then Fact:=1 else Fact:=Fact(n-1)*n
        end;
```

În atribuirea `Fact:=Fact(n-1)*n` am folosit identificatorul `Fact` în două feluri diferite: `Fact:=...` înseamnă că se pregăteşte valoarea ce trebuie returnată de funcția `Fact`, iar `Fact(n-1)` înseamnă un apel (recursiv) al acestei funcții pentru parametrul efectiv `n-1`. Astfel, a calcula `Fact(2)` înseamnă: Este `2=0`? Nu, atunci `Fact(2)` va fi `Fact(1)` înmulțit cu `2`. Dar `Fact(1)` se determină exact la fel. Este `1=0`? Nu, atunci `Fact(1)` va fi `Fact(0)` înmulțit cu `1`. Dar `Fact(0)` se determină exact la fel. În sfârşit, `n` (adică `0`) este egal cu zero, aşa încât obținem `Fact(0)=1`. În acest moment, avem o reîntoarcere la `Fact(1)`, obținând `Fact(1)=1` şi transmițând această valoare lui `Fact(2)`, care o va înmulți cu `2` şi va obține rezultatul `2`. Deci: `Fact(2)=Fact(1)×2=(Fact(0)×1)×2=(1×1)×2=2`.
Propunem cititorului redactarea întregului program.

4. În cele ce urmează vom prezenta un exemplu ceva mai complex. Vom scrie o funcție pentru determinarea celui mai mare divizor comun al `n` numere întregi. Funcția va fi recursivă, conform relației: $CMMDC(x_1,...,x_{n-1},x_n) = CMMDC(CMMDC(x_1,...,x_{n-1}),x_n)$.
Astfel, vom scrie două funcții. Prima va determina cel mai mare divizor comun a două numere întregi `a` şi `b`. Cu cea de a doua, recursivă, se va determina cel mai mare divizor comun al `n` numere întregi, astfel: se va determina `d`=cel mai mare divizor comun al primelor `n-1` numere, apoi se va determina cel mai mare divizor comun al numerelor `d` şi x_n. Vom memora numerele într-un vector.

```
program Calcul_CMMDC_n_numere;
type vector = array[1..10] of Integer;
var a: vector; x, n, i: Integer;
function CMMDC2(a,b: Integer): Integer;
begin
   while a<>b do if a>b then a:=a-b else b:=b-a;
   CMMDC2:=a
end;
function CMMDC(x: vector; n: Integer): Integer;
begin
   if n=2 then CMMDC:=CMMDC2(x[1],x[2])
   else CMMDC:=CMMDC2(CMMDC(x,n-1),x[n])
end;
begin
   Write('Dati numarul de elemente: '); ReadLn(n);
   for i:=1 to n do
     begin Write(' - dati numarul al ',i,'-lea: '); ReadLn(a[i]) end;
   x:=CMMDC(a,n);
   WriteLn('Cel mai mare divizor comun este = ',x);
   ReadLn
end.
```

În programul precedent ne-am jucat atât cu identificatorii de variabile, cât şi cu apelurile dintr-o funcție a unei alte funcții şi, de asemenea, cu recursivitatea. Observăm că, deoarece funcția `CMMDC` apelează funcția `CMMDC2`, aceasta din urmă a fost scrisă înaintea primeia. În blocul principal, `a` este un vector, iar `x` un număr întreg. În cadrul funcției `CMMDC2`, `a` este un număr întreg, iar în cadrul funcției `CMMDC x` este un vector. Pentru datele de intrare `n=3`, `a=(10,14,6)` se va afişa `2`.

5. Funcția `CMMDC2` de mai sus se poate scrie şi în alte două moduri, ambele bazându-se pe **algoritmul clasic al lui Euclid**. Cea de a doua variantă, recursivă, se bazează pe relația matematică: `cmmdc(a,b)=cmmdc(b, a mod b)`.

a) *Varianta iterativă:*

```
function CMMDC2(a,b: Integer): Integer;
var aux: Integer;
begin
        while b<>0 do begin aux:=b; b:=a mod b; a:=t end;
        CMMDC2:=a
end;
```

b) *Varianta recursivă:*

```
function CMMDC2(a,b: Integer): Integer;
begin
        if b=0 then CMMDC2:=a else CMMDC2:=CMMDC2(b, a mod b)
end;
```

6. În continuare, punem în evidență situațiile în care modificările argumentelor se transmit în exteriorul funcției. În această situație, nu putem apela funcțiile decât cu variabile. Să scriem, în două variante, o funcție care să realizeze următorul lucru: să pună în fața şi în spatele unui şir de caractere câte un semn '#', apoi să returneze lungimea şirului nou format. De pildă, pentru şirul 'abc' să se obțină valoarea 5, efectul lateral al funcției fiind transformarea şirului în '#abc#'. Vom observa că în prima din variante nu obținem rezultatul aşteptat, tocmai fiindcă parametrii formali ai funcției nu au fost declarați ca fiind variabili. Însă a doua variantă funcționează corect.

```
1)      function Gresit(S: String): Integer;
        begin
                S:='#'+S+'#'; Gresit := Length(S)
        end;
2)      function Corect(var S: String): Integer;
        begin
                S:='#'+S+'#'; Corect := Length(S)
        end;
```

Să numim *F* oricare din funcțiile anterioare. Un program de test ar putea fi:

```
program Test2;
function F(...):...;
begin
        ......
end;
var S: String;
begin
        S:='abc';
        WriteLn(F(S));
        ReadLn
end.
```

În prima situație putem apela direct funcția `Gresit` prin `Gresit('abc')`. Dar un apel de genul `Corect('abc')` va semnala eroare sintactică!
Când efectuăm un apel în care parametrii formali nu sunt precedați de cuvântul `var`, se spune că are loc un **apel prin valoare**. În celălalt caz, avem un **apel prin referință**. Într-adevăr, în al doilea caz, parametrii formali nu se creează în altă parte, în cadrul funcției, ci *se suprapun* peste cei efectivi.
Deci, pe când în cazul apelului prin valoare comunicarea este doar în sensul de la blocul apelant la cel apelat, în cazul apelului prin referință comunicarea se realizează în ambele sensuri.

Un exemplu reprezentativ pentru apelul prin referință este cazul interschimbării a două variabile. Vom scrie o astfel de funcție a carei valoare returnată nu contează.

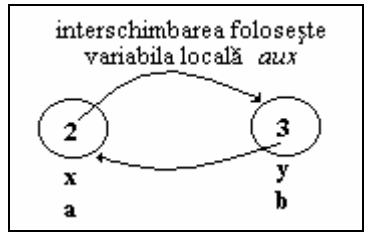
```
function Schimba(var a,b: Integer): Boolean;
var aux: Integer;
begin
   aux:=a;
   a:=b;
   b:=aux
end;
```

Un programul complet folosind această funcție este:

```
{$X+}
program Test3;
function Schimba(...):...
......
```

```
.....
var x,y: Integer; b: Boolean;
begin
   x:=2; y:=3;
   Schimba(x,y); { am putea scrie si b := Schimba(x,y) }
   WriteLn(x:3, y:3);
end.
```

Programul va afişa `3`, apoi `2`. Ce se întâmplă, de fapt? Variabilele globale `x` şi `y` primesc valorile `2`, respectiv `3`. În momentul în care se apelează funcția `Schimba` (care se apelează ca o procedură, datorită opțiunii de compilare de la începutul programului!), nu se creează două noi variabile `a` şi `b`, ci `x` şi `y` primesc încă un nume. Astfel, orice acțiune asupra lui `a` şi `b` se va repercuta şi asupra lui `x` şi `y`.



În concluzie, ca o regulă generală, atunci când se doreşte ca o procedură să modifice valoarea unui argument, acesta se va declara ca *parametru valoare*, altfel ca *parametru variabilă (referință)*. Totuşi, când folosim structuri de date de dimensiuni considerabile, utilizarea parametrului valoare ocupă atât spațiu de memorie, cât şi timp de execuție. Este preferabil, în această situație, declararea parametrului respectiv ca variabil; printr-un comentariu se poate preciza ca procedura să nu îi modifice valoarea.

7. În continuare, vom scrie un program care să afişeze toate numerele prime mai mici sau egale cu un număr `n` dat. Vom scrie, o funcție `EstePrim` care verifică primalitatea unui numar întreg `x`. Această funcție va apela la o funcție proprie de calcul a radicalului unui număr real `x`, pe care o vom numi sugestiv `Radical`. Dacă vrem să dispunem de funcția `Radical` şi în blocul principal, atunci o vom scrie în cadrul său. Singura restricție va fi aceea ca funcția `Radical` să apară înainte de funcția `EstePrim`.
Nu vom proceda, însă, aşa, ci vom declara şi defini funcția `Radical` în cadrul (restrâns) al funcției `EstePrim`. Astfel, nu vom putea dispune în altă parte de această funcție! Acest mod de realizare a programelor se numeşte *modularizat* şi vom reveni asupra subiectului.

```
program NumerePrime;
function EstePrim(x: Integer): Boolean;
   function Radical(xx: Real): Real;
   const eps=0.0001;
   var an,an1: Real;
   begin
     an1:=1; an:=(1+xx)/2;
     while abs(an-an1)>eps do begin an1:=an; an:=(an+xx/an)/2 end;
     Radical:=an
   end; {Radical }
var d,rad: Integer; prim: Boolean;
begin {functia EstePrim}
  if x mod 2 = 0 then if x=2 then prim:=True else prim:=False
  else
    begin
      d:=3; prim:=True; rad:=Trunc(Radical(x));
      while (d<=rad) and prim do
            if x mod d = 0 then prim:=False else d:=d+2
    end;
```

```
EstePrim:=prim
end; {EstePrim }
var n,i: Integer;
begin { Program principal }
  WriteLn('Determinare numere prime'); Write('Dati n = '); ReadLn(n);
  for i:=2 to n do if EstePrim(i) then Write(i:3,', '); ReadLn
end.
```

Modularizarea. Domenii de vizibilitate a identificatorilor

După cum am spus, un program în Pascal este un **bloc** (modul). Un bloc, la rândul său, poate conține alte blocuri (module) ş.a.m.d., adică ele pot fi imbricate unele în altele. Fiecare bloc va fi desemnat prin identificatorul funcției, procedurii sau al programului corespunzător.

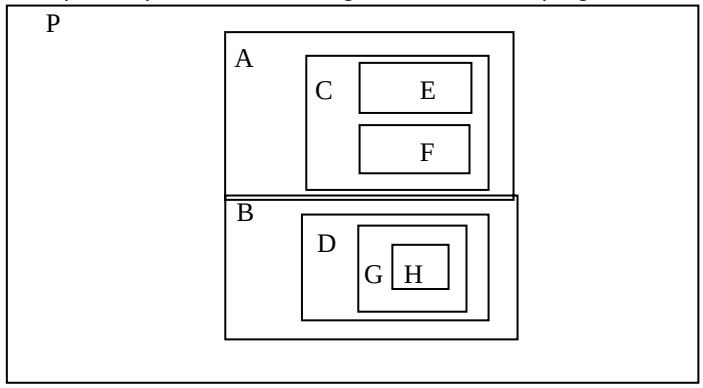
Există trei reguli de valabilitate sau vizibilitate a identificatorilor:

- În cadrul unui bloc, un identificator poate fi declarat o singură dată.

Un acelaşi identificator poate fi declarat în blocuri diferite. În acest caz se aplică regula:

- Natura unui identificator `id` care apare într-o instrucțiune se stabileşte căutând cel mai interior bloc ce include atât instrucțiunea, cât şi declararea lui `id`.
- Nu se poate folosi un identificator în exteriorul blocului în care a fost declarat.

Pentru a exemplifica aplicarea acestor reguli fie schema de program modularizat:



Putem vorbi de o ierarhizare a blocurilor pe nivele:

Bloc	Nivel
P	0
A,B	1
C,D	2
E,F,G	3
H	4

Domeniile de vizibilitate (valabilitate) ale identificatorilor sunt:

Identificatorii declarați în blocul:	sunt accesibili în blocul:
P	P, A, B, C, D, E, F, G, H
A	A, C, E, F
B	B, D, G, H
C	C, E, F
D	D, G, H
E	E
F	F
G	G, H
H	H

De exemplu, o variabilă declarată în blocurile `B` şi `G` şi apelată în blocul `H` va avea tipul precizat în declarația conținută de blocul `G`.

Proceduri

Conceptul de procedură este familiar cititorului care a parcurs cu atenție lecțiile primului capitol.

După cum am spus deja, **procedura** este o funcție particulară, ea nu returnează valori. Astfel, identificatorul ei nu poate reprezenta decât procedura însăşi, (nu şi vreo valoare returnată). O procedură va fi apelată prin identificatorul ei urmat de lista parametrilor efectivi.

Declararea şi definirea unei proceduri se face astfel:

```
procedure id_proc (lista_de_parametri_formali);
declaraţii locale;
begin
   instrucţiuni
end;
```

Apelul se va realiza prin `id_proc(lista_de_param_efectivi)`. Pot exista atât parametri variabili (referință), cât şi parametri valoare. Pot exista proceduri fără parametri. La o procedură contează efectele laterale şi valorile returnate, prin intermediul parametrilor referință.

În lecțiile anterioare am utilizat proceduri predefinite. Astfel, de exemplu `WriteLn` are parametrii formali transmişi prin valoare. Procedura `ReadLn` are parametrii transmişi prin referință, tocmai pentru că ei trebuie să se modifice (actualizându-se la valorile citite de la tastatură).

Ideii de *programare structurată* i se adaugă acum cea de **programare modularizată**, care permite realizarea unor programe foarte flexibile şi eficiente, bine structurate. Mai târziu vom vorbi despre **programarea orientată pe obiecte**, care face ca şi datele programelor să fie bine structurate.

2. Aplicații ale subprogramelor

1. Admiterea în liceu.

Pentru fiecare din etapele importante ale programului, vom defini câte o procedură. Procedurile vor fi apelate, pe rând, conform selecției instrucțiunii `case` din blocul principal. Apelurile se vor realiza într-un ciclu. Acesta se va termina când se va opta pentru opțiunea '`s`'.

```
program AdmitereLiceu;
{ varianta cu "record" si proceduri}
uses Crt;
type elev = record nume: String; nota1,nota2,media: Real end;
type vector=array[1..20] of elev;
var elevii: vector; NrElevi: Integer; optiune: Integer; date_introduse:
Boolean;
procedure IntroducereDateElevi(var E:vector; var n: Integer);
var i: Integer;
begin
  if not date_introduse then optiune:=1
  else begin
         WriteLn('Atentie !  Aveti datele deja introduse !');
         WriteLn('Doriti reintroducerea lor ? (1=da, 2=nu)');
         ReadLn(optiune);
       end;
  if optiune=1 then
     begin
       Write('Dati numarul de elevi : '); ReadLn(n);
       for i:=1 to n do
         with E[i] do
           begin
             WriteLn('Dati numele si prenumele !'); ReadLn(nume);
             WriteLn('Dati notele !'); ReadLn(nota1, nota2);
```

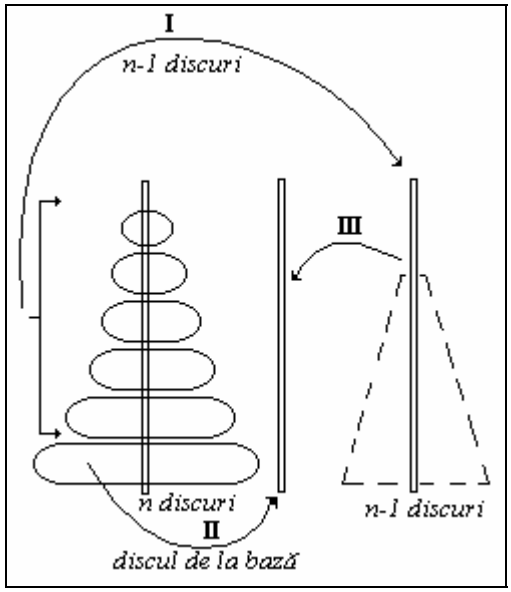
```
            media:=(nota1+nota2)/2
          end;
       date_introduse:=True
     end
end;
procedure ListareElevi(E: vector; n: Integer);
var i,j: Integer; numele: String;
begin
  for i:=1 to n do
      with E[i] do
        begin { se completeaza numele sau cu spatii la dreapta }
          numele:=nume;
          for j:=1 to 40-Length(nume) do numele:=numele+' ';
          WriteLn(i:2,'. ',numele,' : ', nota1:6:2,
                  nota2:6:2, media:6:2)
        end
end;
procedure OrdonareDupaMedii(n: Integer; var E: vector);
var i,j: Integer; aux: elev;
begin
  if not date_introduse then
     WriteLn('Introduceti, mai intai datele elevilor!')
  else
    begin
      for i:=1 to Pred(n) do
          for j:=Succ(i) to n do { comparatia se face dupa medii }
              if E[i].media < E[j].media then
                  { are loc interschimbarea elevilor }
                begin aux:=E[i]; E[i]:=E[j]; E[j]:=aux end;
      ListareElevi(E,n)
    end
end;
procedure OrdonareAlfabetica(n: Integer; var E: vector);
var i,j: Integer; aux: elev;
begin
  if not date_introduse then
     WriteLn('Introduceti, mai intai datele despre elevi !')
  else
    begin
      for i:=1 to Pred(n) do
          for j:=Succ(i) to n do { comparatia se face dupa nume }
              if E[i].nume > E[j].nume then
                 { are loc interschimbarea elevilor }
                 begin aux:=E[i]; E[i]:=E[j]; E[j]:=aux end;
      ListareElevi(E,n)
    end
end;
procedure Meniu;
begin
  ClrScr; WriteLn('ORDONARE ELEVI LA CONCURS'); WriteLn;
  WriteLn('0 = oprire program');
  WriteLn('1 = introducere date elevi');
  WriteLn('2 = listare dupa medii');
  WriteLn('3 = listare alfabetica');
  WriteLn; Write('Optiunea dvs. = '); ReadLn(optiune);
end;
begin
  date_introduse:=False;
  repeat
    Meniu;
    case optiune of
         1: IntroducereDateElevi(elevii, NrElevi);
         2: OrdonareDupaMedii(NrElevi, elevii);
         3: OrdonareAlfabetica(NrElevi, elevii)
    end;
```

```
    WriteLn; WriteLn('Apasati <Enter> pentru continuare !'); ReadLn
  until optiune=0
end.
```

2. Simplificarea unei fracții. *S*ă se simplifice o fracție a/b.
Vom proceda în felul următor: dându-se două numere a şi b, vom verifica dacă b este diferit de zero sau nu. În caz că nu, se va afişa un mesaj de eroare. Altfel, dacă a nu este nul, se va determina, cu o funcție corespunzătoare, cel mai mare divizor comun a lui a şi b. Fie acesta c. Rezultatul va fi dat sub forma u/v, unde u şi v se determină împărțind pe a şi b la c. Dacă v=1, atunci vom afişa doar pe u.

```
program SimplificareaUneiFractii;
function cmmdc(a,b: Integer): Integer;
begin
  while a<>b do if a>b then a:=a-b else b:=b-a;
  cmmdc:=a
end;
procedure Simplifica(a,b: Integer; var u,v: Integer);
var c: Integer;
begin
  c:=cmmdc(a,b); u:=a div c; v:=b div c
end;
var a,b,u,v: Integer;
begin
  WriteLn('Simplificare'); WriteLn('Dati a si b !');
  ReadLn(a,b);
  if b=0 then WriteLn('Fractie inexistenta !')
  else if a=0 then WriteLn('Rezultat: 0')
       else begin
              Simplifica(a,b,u,v);
              if v=1 then WriteLn('Rezultat: ',u)
              else WriteLn('Rezultat: ',u,'/',v)
            end;
  ReadLn
end.
```

3. Turnurile din Hanoi. Se spune că în Vietnamul antic, în Hanoi, erau trei turnuri, pe unul din ele fiind puse, în ordinea descrescătoare a diametrelor lor, mai multe (opt)_discuri de aur. Din motive obiective, nişte călugări, care le aveau în grijă, trebuiau să aşeze discurile pe cel de-al doilea turn, în aceeaşi ordine. Ei puteau să folosească, eventual, turnul al treilea, deoarece, altfel discurile nu ar fi avut stabilitate. Discurile puteau fi mutate unul câte unul. De asemenea, se renunța din start la ideea de a aşeza un disc mai mare peste unul mai mic, deoarece exista riscul ca să se strice discurile, din cauza diferenței mari de masă.
Deşi, aparent simplă, după câteva încercări, cu un prototip în față, cititorul va constata că problema nu e banală. Însă este posibilă o rezolvare optimă a ei (cu numai 2^n-1 mutări, deci 255, pentru n=8), folosind tehnica recursivă "*divide et impera*", asupra căreia vom reveni în capitolul următor.
Problema este de a muta n discuri de la turnul 1 la turnul 2. Pentru a o rezolva, să vedem cum se mută, în general, m discuri de la un turn p la un turn q. Se mută primele m-1 discuri de pe p pe r, r fiind turnul auxiliar, apoi singurul disc rămas pe p (discul cel mai mare) se mută de pe p pe q, după care cele m-1 discuri sunt mutate de pe r pe q. Fireşte, mutarea celor m-1 discuri de la p la r şi de la r la q se realizează la fel, deci recursiv. Mutarea primelor m-1 discuri este corectă, deoarece existența discurilor de diametre mai mari, la bazele celor trei turnuri nu afectează cu nimic mutările discurilor mai mici.
În cazul limită m=1, avem doar o mutare a discului din vârful turnului p spre q.



Programul de mai jos soluționează (cu animație) problema descrisă, pentru `n=8`. Procedura de bază este `Han`, iar celelalte proceduri sunt pentru mişcarea discului curent. Există şi două proceduri cu structură inedită. Ele sunt scrise în limbaj de asamblare. Folosind întreruperea `10h`, realizează ascunderea, respectiv reafişarea cursorului text.

```
program TurnurileDinHanoi;
uses Crt;
const Pauza=10; forma= #219; Virf: array [1..3 ] of Byte=(13,22,22);
procedure HideCursor; assembler; { ascunde cursorul pilpâitor, in modul
text }
asm MOV AX,$0100; MOV CX,$2607; INT $10 end;
procedure ShowCursor; assembler; { reafiseaza cursorul }
asm MOV AX,$0100; MOV CX,$0506; INT $10 end;
function ColTija (tija : Byte) : Byte; {stabileste coloana unei tije}
begin ColTija := 24*tija-8 end;
procedure MutaDreapta (disc, tija1, tija2 : Byte);
var i,k: Byte;
begin
     for i := ColTija(tija1)-disc to Pred(ColTija(tija2)-disc) do
         begin
           Delay(Pauza); if KeyPressed then Halt(1);
           GoToXY(i,3); for k:=0 to 2*disc do Write(' ');
           GoToXY(i+1,3); for k:=0 to 2*disc do Write(forma)
         end
end;
procedure MutaStanga (disc, tija1, tija2 : Byte);
var i,k: Byte;
begin
     for i := ColTija(tija1)-disc downto Succ(ColTija(tija2)-disc) do
         begin
           Delay(Pauza); if KeyPressed then Halt(1);
           GoToXY(i,3);
           for k:=0 to 2*disc do Write(' ');
           GoToXY(i-1,3);
           for k:=0 to 2*disc do Write(forma)
         end
end;
procedure Coboara (disc, tija : Byte);
var i,k:  Byte;
```

```
begin
  for i := 3 to Pred(Virf[tija]-1) do
    begin
      Delay(Pauza); if KeyPressed then Halt(1);
      GoToXY(ColTija(tija)-disc,i);
      for k:=0 to 2*disc do Write(' ');
      GoToXY(ColTija(tija)-disc,i+1);
      for k:=0 to 2*disc do Write(forma)
    end;
  Dec(Virf[tija])
end;
procedure Ridica (disc, tija : Byte);
var i,k: Byte;
begin
  for i := Virf[tija] downto 4 do
    begin
      Delay(Pauza); if KeyPressed then Halt(1);
      GoToXY(ColTija(tija)-disc,i);
      for k:=0 to 2*disc do Write(' ');
      GoToXY(ColTija(tija)-disc,i-1);
      for k:=0 to 2*disc do Write(forma)
    end;
  Inc(virf[tija])
end;
procedure Muta(disc, tija1, tija2 : Byte);
begin
  Ridica(disc,tija1);
  if (tija1 < tija2) then
     MutaDreapta(disc,tija1,tija2)
  else
     MutaStanga(disc,tija1,tija2);
  Coboara(disc,tija2)
end;
procedure Han(n, tija1, tija2, tija3 : Byte);
begin
  if (n = 1) then Muta(1,tija1,tija2)
  else
    begin
      Han(n-1,tija1,tija3,tija2);
      Muta(n,tija1,tija2);
      Han(n-1,tija3,tija2,tija1)
  end
end;
procedure Initializari;
var k,disc: Byte;
begin
  HideCursor; ClrScr;
  for disc:=1 to 9 do
      begin
        GoToXY(ColTija(1)-disc,Virf[1]+disc-1);
        for k:=0 to 2*disc do
        Write(forma);
      end
end;
begin { PROGRAM }
  Initializari; GoToXY(28,1); WriteLn(' ~Turnurile din Hanoi~ ');
  Han(8,1,2,3); ShowCursor
end.
```

Citorii care nu cunosc elemente de limbaj de asamblare vor considera procedurile `ShowCursor` şi `HideCursor` (şi altele asemănătoare) ca atare. Mai puțin importantă este înțelegerea modului cum sunt ele realizate şi mai mult ce execută ele.

În cadrul unui text sursă Pascal, pot apare secvențe de limbaj de asamblare, încadrate de cuvintele **asm** şi **end**. Dacă o procedură este scrisă în întregime în limbaj de asamblare, se va preciza aceasta prin cuvântul rezervat **assembler**. Aşa am procedat şi noi.

Procedura **Halt** (apărută în program) va determina oprirea forțată a programului.

3. Variabile inițializate. Variabile de tip funcție sau procedură

Variabile inițializate

Un element surpriză în programul anterior (din secțiunea 2) îl constituie declarația:

```
const Virf: array [1..3 ] of Byte=(13,22,22);
```

Este vorba despre definirea unei variabile (nu a unei constante!), care se inițializează la o anumită valoare. Astfel, când vrem să inițializăm vectori se vor scrie elementele acestuia într-o listă încadrată de două paranteze. Când avem matrice, avem o listă de liste, cuprinse între paranteze, iar în cazul variabilelor simple vom avea valorile, pur şi simplu. Un caz special îl constituie înregistrările, asupră cărora vom reveni atunci când ne vom reîntâlni cu această problemă.

Propunem cititorului să îmbunătățească aplicația 2 din secțiunea 2, astfel încât meniul să fie mai atractiv. Comenzile meniului să aibe asociate nişte explicații. Comenzile şi explicațiile asociate să fie memorate direct în variabile vectori inițializați.

Variabile de tip subprogram

Pentru a determina minimul, respectiv maximul, dintr-un vector de elemente oarecare se procedează similar, singura deosebire fiind de un semn ("<" sau ">" după caz). Astfel, să unificăm cele două relații de inegalitate sub numele unei funcții generice `Relatie`, care având două argumente numere întregi `x` şi `y` să întoarcă `True` sau `False`, după cum între `x` şi `y` este stabilită sau nu relația dorită.

Limbajul Pascal ne permite să folosim pe `Relatie` ca pe o variabilă de tip funcție şi chiar ca parametru în apelul unui subprogram. Determinarea minimului şi maximului dintr-un vector se va face conform programului de mai jos, în care trebuie să observăm prezența opțiunii de compilare `{$F+}`, care ne permite să folosim variabile de tip subprogram.

```
{$F+}
program VariabileSubprogram;
uses Crt;
type functie_de_comparatie = function(x,y: Integer):Boolean;
     vector = array[1..10] of Integer;
var x: vector; i,n, optim: Integer; Relatie: functie_de_comparatie;
function MaiMic(x,y: Integer): Boolean;
begin MaiMic := x<y end;
function MaiMare(x,y: Integer): Boolean;
begin MaiMare := x>y end;
begin
     WriteLn('Determinare minim sau maxim');
     Write('n = '); ReadLn(n);
     for i:=1 to n do begin Write('x[',i,']='); ReadLn(x[i]) end;
     WriteLn('Ce optim doriti sa determinam ?');
     WriteLn('m = minim, M = maxim');
     if ReadKey='m' then Relatie:=MaiMic else Relatie:=MaiMare;
     optim:=x[1];
     for i:=2 to n do if Relatie(x[i],optim) then optim:=x[i];
     WriteLn('Optimul determinat este = ',optim); ReadLn
end.
```

Asupra acestui tip special de variabile vom reveni în lecția 11, unde vom prezenta o procedură de reprezentare grafică a unei funcții oarecare, al cărei corp este prezent în textul programului.

4. Recursivitate

Noțiunea de recursivitate (sau recursie) vă este deja familiară, am folosit-o până acum de mai multe ori.

În continuare ne vom ocupa de:

- condiția de consistență a unei definiții recursive;
- utilizarea stivei în recursivitate;
- tipurile de recursivitate: directă şi indirectă.

Condiția de consistență a unei definiții recursive

Să considerăm definiția recursivă de mai jos (*funcția lui Ackermann*):

$$A(m,n)=\begin{cases} n+1, & daca \quad m=0, \\ A(m-1,1), & daca \quad n=0, \\ A(m-1,A(m,n-1)), & altfel. \end{cases}$$

În limbajul Pascal, această funcție se va scrie:

```
function A(m,n: Integer);
begin
        if m=0 then A:=n+1
        else if  n=0 then A:=A(m-1,1) else A:=A(m-1,A(m,n-1))
end;
```

Să determinăm `A(2,2)`. Vom aplica succesiv definiția până la identificarea unor argumente pentru care valoarea funcției `A` se poate calcula. Apoi vom substitui valoarea calculată în locul celui mai interior argument de pe poziția a doua, după care vom relua aplicarea definiției, dacă numele funcției nu mai apare. Avem succesiv:

```
A(2,2)=A(1,A(2,1))=A(1,A(1,A(2,0)))=A(1,A(1,A(1,1)))=A(1,A(1,A(0,A(1,0))))=A(1,A
(1,A(0,A(0,1))))=A(1,A(1,A(0,2)))=A(1,A(1,3))=A(1,A(0,A(1,2)))=A(1,A(0,A(0,A(1,1
))))=A(1,A(0,A(0,A(0,A(1,0)))))=A(1,A(0,A(0,A(0,A(0,1)))))=A(1,A(0,A(0,A(0,2))))
=A(1,A(0,A(0,3)))=A(1,A(0,4))=A(1,5)=A(0,A(1,4))=A(0,A(0,A(1,3)))=A(0,A(0,A(0,A(
1,2))))=A(0,A(0,A(0,A(0,A(1,1)))))=A(0,A(0,A(0,A(0,A(0,A(1,0))))))=A(0,A(0,A(0,A
(0,A(0,A(0,1))))))=A(0,A(0,A(0,A(0,A(0,2)))))=A(0,A(0,A(0,A(0,3))))=A(0,A(0,A(0,
4)))=A(0,A(0,5))=A(0,6)=7.
```

Observăm tendința de identificare a unor valori direct calculabile, urmată de utilizarea lor în vederea obținerii valorilor căutate.

Astfel, o definiție recursivă trebuie să satisfacă următoarea condiție: *valoarea funcției trebuie să fie ori direct calculabilă, ori calculabilă cu ajutorul unei valori direct calculabile*. Această condiție se numeşte **condiția de consistență**, iar ea este verificată atât de funcția lui *Ackermann*, cât şi de funcția recursivă cu ajutorul căreia calculasem factorialul unui număr.

Un exemplu de funcție inconsistentă este următoarea:

```
function Inconsistent(n: Integer);
begin
   if n=0 then Inconsistent:=1
   else Inconsistent:=n*Inconsistent(n+1)
end;
```

Utilizarea stivelor în recursivitate

Mai întâi trebuie să precizăm că, în informatică, prin stivă se înțelege ceea ce se înțelege şi în viața de zi cu zi. Deocamdată vom considera o stivă ca fiind un vector asupra căruia se pot face operațiile: adăugarea unui element, după ultimul element (indicat de un număr `n`), şi eliminarea

ultimului element (al n-lea) din vector. Vectorul ar putea fi asemuit unei stive de cărți: putem pune o carte nouă peste celelalte deja existente şi putem să luăm cartea de deasupra.
Atunci când, de exemplu, se doreşte calcularea valoarii 3!, se memorează într-o stivă, inițial vidă, acest număr 3, necesar pentru a fi înmulțit cu 2!. Procesul continuă până la 0!, când are loc eliminarea, pe rând, a elementelor din stivă, simultan cu câte o înmulțire.

```
function Fact(n: Integer);
begin
       if n=0 then Fact:=1 else Fact:=Fact(n-1)*n
end;
```

Mediul Turbo-Pascal dispune de o stivă proprie, cu o anumită dimensiune. Depăşirea acesteia se face cu opțiunea de compilare {$S+}. Ea este implicită. Când lucrăm cu apeluri recursive ce încarcă foarte mult stiva, dar avem condiția de consistență a recursiei îndeplinită, este bine să folosim opțiunea de compilare {$S-}.
Noi înşine putem simula recursivitatea, implementând o structură proprie de stivă. Următorul program calculează factorialului unui număr, folosind o iterație asupra unei stive, iterație ce simulează recursivitatea.

```
program SimulareRecursivitate;
const max=20;
type stiva=record
             x: array[1..max] of Integer; n: Integer
          end;
procedure InitStiva(var S: stiva);
begin S.n:=0 end;
function EsteGoalaStiva(var S: stiva): Boolean;
{ folosim "var" pentru economie }
begin EsteGoalaStiva:=S.n=0 end;
procedure PuneInStiva(var S: stiva; elem: Integer);
begin
     with S do
          if n=max then WriteLn('Stiva plina !')
          else begin Inc(n); x[n]:=elem end
end;
procedure ScoateDinStiva(var S: stiva; var elem: Integer);
begin
     with S do
          if n=0 then WriteLn('Stiva goala !')
          else begin elem:=x[n]; Dec(n) end
end;
function Factorial(n: Integer): Integer;
var S: stiva; F: Integer;
begin
     InitStiva(S); F:=1;
     repeat
       PuneInStiva(S,n); Dec(n)
     until n=0;
     F:=1;
     repeat
        ScoateDinStiva(S,n); F:=F*n
     until EsteGoalaStiva(S);
     Factorial:=F
end;
var n: Integer;
begin
     WriteLn('Calcul factorial - simulare recursivitate');
     Write('Dati n = '); ReadLn(n);
     WriteLn(n,'! = ',Factorial(n)); ReadLn
end.
```

Asupra stivelor vom reveni în lecția următoare, odată cu prezentarea tehnicii de programare "*back-tracking*", precum şi în lecția 8, unde vor fi prezentate structurile *dinamice* de date.

Structurile de date folosite până acum sunt *statice*. Astfel, noi nu putem calcula, cu programul de mai înainte, factorialul unui număr mai mare decât constanta `max`.

Recursivitate directă şi indirectă

- Toate subprogramele recursive prezentate până acum se autoapelau în mod *direct*, motiv pentru care se spune că s-a folosit **recursivitatea directă**.
- Există, însă, şi situații când un subprogram `P` va apela la el însuşi din cadrul altui subprogram `Q`, apelat de `P`. Asfel, cele două subprograme se apelează reciproc. Conform regulilor de sintaxă şi topică ale limbajului, cum `P` apelează pe `Q`, înseamnă că `Q` va trebui să apară înaintea lui `P`. Or şi `Q` apelează pe `P`, iar pentru a soluționa acest impas, se declară unul din subprograme înaintea celuilalt: se scrie doar antetul său, urmat de cuvântul rezervat **`forward`**. Apoi, când se defineşte subprogramul declarat înainte ("forward"), în scrierea antetului se poate renunța la lista parametrilor.

Spunem că avem de a face cu o **recursivitate indirectă** sau **încrucişată**.

Iată un exemplu:

```
program RecursivitateIndirecta;
function F(x: Integer): Integer; forward;
function G(x: Integer): Integer;
begin if x=0 then G:=1 else G:=G(x-1)+F(x-1) end;
function F;
begin if x=0 then F:=1 else F:=F(x-1)+G(x-1) end;
begin
       WriteLn(F(5));
       ReadLn
end.
```

Programul va afişa valoarea `32`. Exemplul prezentat este pur teoretic, noi vom încerca să evităm utilizarea acestui gen de recursivitate, deoarece de multe ori duce la programe mai greu de urmărit.

Exerciții

Exercițiile următoare se cer a fi rezolvate folosind subprograme recursive:

1. Să se calculeze suma `S(n)=1+3+5+...+(2n-1).`
2. Să se calculeze produsele $P_1(n)=1\times4\times7\times...\times(3n-2)$ şi $P_2(n)=2\times4\times6\times...\times(2n)$.
3. Să se determine produsul componentelor unui vector.
4. Să se inverseze un şir de caractere.
5. Se consideră declarația de tip: `type vector=array[1..20] of Integer`. Să se verifice apartenența unui element la un vector `var x: vector`.
6. Să se verifice dacă un vector conține cel puțin un număr negativ în primele `n` poziții.
7. Să se afişeze conținutul unui vector.
8. Să se inverseze un vector.

5. Aplicație: evaluator de expresii algebrice

Descrierea algoritmului

În cele ce urmează ne propunem să realizăm un program care să citească expresia oricărei funcții şi să o poată tabela, adică, dându-i-se un număr de puncte dintr-un interval, să afişeze valorile funcției în fiecare din acele puncte. Expresia funcției va fi memorată într-un şir de caractere. Ea va putea conține operații şi funcții elementare (adunare, scădere, înmulțire, împărțire, ridicare la putere, `exp`, `ln`, `abs`, radical, sin şi cos)). Pentru a putea utiliza inclusiv funcții "cu acoladă", vom folosi o convenție din limbajul *BASIC*.

Astfel, să considerăm următoarele două funcții:

$$f(x)=\begin{cases} x\cdot\cos(1/x), & \text{dac\ } x\neq 0 \\ 0, & \text{dac\ } x=0. \end{cases}$$

şi

$$f(x)=\begin{cases} x^2, & \text{dac\ } x\geq 0 \\ x-\sin(x\cdot 2), & \text{altfel} \end{cases}$$

Prima funcție va fi dată sub forma: `(x#0)*(x*cos(1/x))+(x=0)*0`; iar a doua sub forma: `(x>0)*(x^2)+(x<0)*(x-sin(x*2))`;
Deci se vor folosi simbolurile "<" (mai mic sau egal), ">" (mai mare sau egal), "=" (egal) şi "#" (diferit), precum şi convenția din limbajul BASIC, anume că expresiile relaționale se evaluează la `1`, dacă sunt adevărate, respectiv la `0`, dacă sunt false.

Evaluarea funcției într-un punct `x` are la bază următoarea idee:
- de fiecare dată când în expresia funcției apare litera "`x`", aceasta va fi înlocuită cu valoarea lui `x`;
- se vor folosi două stive: una a operanzilor (`Opd` - cu vârful `top1`) şi una a operatorilor (`Op` - cu vârful `top2`);
- până la întâlnirea unei paranteze închise, care ar semnala închiderea unui bloc, cu stivele se lucrează în maniera următoare:

* de fiecare dată când este nevoie, se pune în stiva `Opd` operandul întâlnit;

* se pune în stiva `Op`, la început, '`(`', apoi, pe rând, operatorii intâlniți, doar dacă în vârful stivei nu este un operator de prioritate mai mare sau egală; în caz contrar, se scot operatorul din vârful stivei şi cei doi operanzi din vârful stivei operanzilor, punându-se în stiva operanzilor rezultatul obținut prin aplicarea operatorului asupra celor doi operanzi;

- procedeul se repeta pâna când stiva operantorilor este vidă şi nimic nu mai e de depus; în acest moment, stiva operanzilor va conține exact valoarea funcției în punctul respectiv.

Să exemplificăm funcționarea algoritmului anterior pentru expresia "`x+abs(x-2*3)`" şi valoarea lui `x=5`.

În primul rând, expresiei i se va adăuga încă o paranteză "`)`", după ce i-au fost înserate spații între *cuvinte*. Apoi, se desparte în cuvinte expresia, memorând cuvintele într-un vector. Prin "*cuvânt*" se va înțelege orice element distinctiv, adică numerele, variabila ("`x`"), operatorii şi funcțiile, parantezele.
Astfel, pentru expresia din exemplul nostru, vom obține următorul vector de lungime 11:

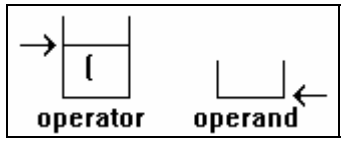
x	+	abs	(	x	-	2	*	3	)	)

Lucrul cu stiva operatorilor depinde de un tabel de priorități care se asociază diferitelor operații:

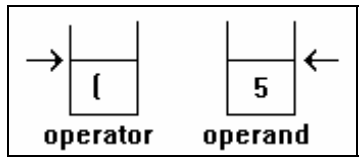
Tabelul priorităților

(	)	+	-	*	/	^	#	=	<	>	cos	sin	ln	exp	abs	rad
0		1		2		3	4				5					

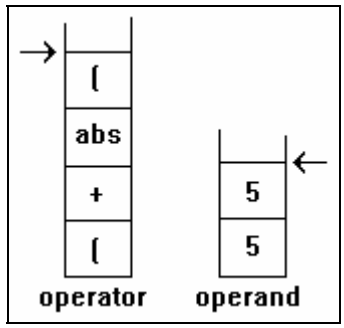
La început, stivele arată astfel:



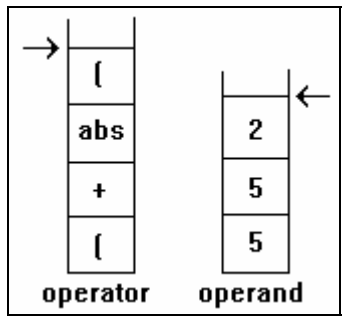
Prima componentă a vectorului nu este număr, ci este "x". Deci se evaluează x, adică în stiva operanzilor se introduce valoarea sa, 5:



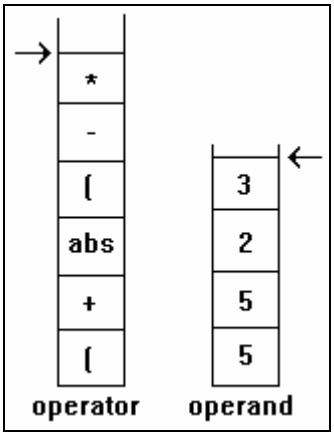
Apoi "+" se introduce în stiva operator, prioritatea lui "+" fiind 1, mai mare decât 0, care e prioritatea lui "(", simbolul din vârful stivei operatorilor. Urmează "abs", care are prioritatea 5, deci mai mare decât a lui "+", aşadar se introduce tot în stivă. Paranteza "(" se va pune şi ea în stivă. Ea marchează începerea unui nou bloc. E rândul lui "x" să fie pus în stiva operanzilor, deci se va pune 5, iar stivele vor arăta acum astfel:



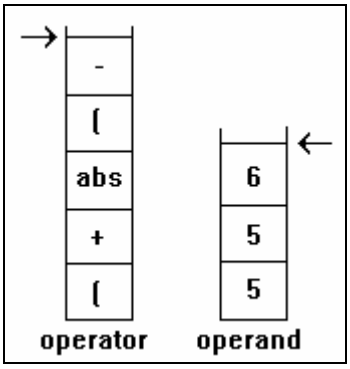
Urmează "-", de prioritate 1 ≥ prioritate('('), deci se pune şi el în stiva operatorilor. În fine, se trece 2 în stiva operanzilor şi, acum, când stivele arată astfel:



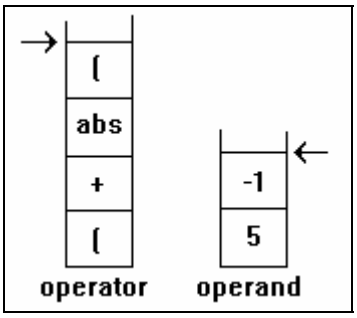
soseşte operatorul "*", prioritar față de "-". Deci se va introduce în stiva operatorilor, iar numărul 3 în stiva operanzilor:



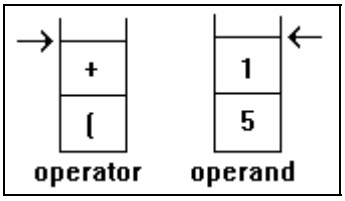
Urmează "`)`", deci se aplică "`*`" asupra lui `3` şi `2`, punându-se rezultatul `3*2=6` în stiva operanzilor:



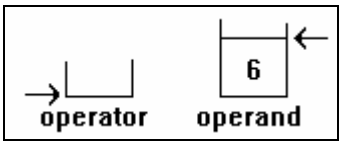
Paranteza ")" închide blocul, deci se aplică "**-**" lui 5 şi 6. Se obține:



S-a terminat cu blocul "`(x-2*3)`", deci se elimină "`(`" şi apoi aplicăm "`abs`" asupra vârfului stivei operanzilor, iar "`abs`" fiind operator unar, se va scoate un element din stiva operanzilor (`-1`) şi se va introduce înapoi tot unul (`|-1|=1`).
Stivele vor arăta astfel:



Se adună `1` cu `5`, se scoate "`+`", apoi "`(`", iar stivele finale sunt:



Acum, stiva operanzilor conține valoarea funcției date, în punctul `x=5`, adică **6**.

Protecția la erori

Programul care implementează algoritmul descris anterior foloseşte o serie de operații şi funcții aritmetice elementare, care au fost rescrise pentru a proteja programul de eventualele erori de genul:

- adunarea, scăderea, înmulțirea, împărțirea a două numere foarte mari (motiv pentru care a fost definită o constantă `infinit`);
- extragerea rădăcinii pătrate dintr-un număr negativ
- calcularea logaritmului natural a unui număr negativ;

Aceste operații, împreună cu cele relaționale (implementate de funcțiile booleene `MaiMic`, `MaiMare` etc. şi de `DifInf` (care verifică dacă un număr este diferit de infinit sau nu)) au fost scrise într-un fişier separat, numit `OPERATII.INC`. Acest fişier se include în textul programului principal prin directiva de compilare **`{$I nume_fis }`**.

Iată fişierul `OPERATII.INC`:

```
{ OPERATII.INC }
const infinit: Real = 30000; epsi: Real=0.0001;
const OperatiiBinare: set of Char = ['+','','*','/','^',
                                     '<','>','=','#'];
      OperatiiUnare: set of Char = ['s','c','a','r','e','l'];
function DifInf(x: Real): Boolean;
begin DifInf := Abs(infinit-Abs(x)) > infinit / 2 end;
function Logaritm(x: Real): Real;
begin
     if (x>epsi) and DifInf(x) then Logaritm:=Ln(x)
     else Logaritm:=infinit {-}
end;
function Exponential(x: Real): Real;
begin
     if DifInf(x) then Exponential:=exp(x) else Exponential:=infinit
end;
function Inmultit(x,y: Real): Real;
begin
     if (Abs(x)<epsi) or (Abs(y)<epsi) then Inmultit:=0
     else if DifInf(x) and DifInf(y) then Inmultit:=x*y
          else Inmultit:=infinit
end;
function Putere(x,y: Real): Real;
var i,yi:integer; p:real;
```

```
begin
  if x=0 then Putere:=0
     else if y=0 then Putere:=1
          else
             if (x=infinit) or (y=infinit) then Putere:=Infinit
             else
               if y=Int(y) then
                  begin
                    yi:=Trunc(y); p:=1;
                    for i:=1 to yi div 2 do p:=p*x; p:=p*p;
                    if odd(yi) then p:=p*x; Putere:=p
                  end
               else Putere:=Exponential(Inmultit(y,Logaritm(x)))
end;
function Egal(x,y: Real): Real;
begin if x=y then egal:=1 else egal:=0 end;
function Diferit(x,y: Real): Real;
var df: Integer;
begin if x=y then df:=0 else df:=1; diferit:=df {1-egal(x,y)} end;
function MaiMic(x,y: Real): Real;
begin if x<y then maimic:=1 else maimic:=0 end;
function MaiMare(x,y: Real): Real;
begin if x>y then maimare:=1 else maimare:=0 end;
function Plus(x,y: Real): Real;
begin if DifInf(x) and DifInf(y) then Plus:=x+y else Plus:=infinit end;
function Minus(x,y: Real): Real;
begin if DifInf(x) and DifInf(y) then Minus:=x-y else Minus:=infinit
end;
function Impartit(x,y: Real): Real;
begin if abs(y)>epsi then Impartit:=x/y else Impartit:=infinit
end;
function Sinus(x: Real): Real;
begin if DifInf(x) then  Sinus:=sin(x) else Sinus:=infinit end;
function Cosinus(x: Real): Real;
begin if DifInf(x) then Cosinus:=cos(x) else Cosinus:=infinit end;
function Modul(x: Real): Real;
begin if DifInf(x) then Modul:=abs(x) else Modul:=infinit end;
function Radical(x: Real): Real;
begin if DifInf(x) and (x>epsi) then Radical:=Sqrt(x) else
Radical:=infinit end;
```

Textul programului

Programul complet pentru evaluarea unei funcții într-un punct este dat mai jos. O expresie este memorată sub forma unei structuri de tip înregistrare:

```
{$X+}
program EvaluatorAlgebric;
uses Crt;
const max=30;
type functie = record
                 expresie: String;
               {expresia functiei = un sir de caractere}
                 vector: array[1..max] of String[10];
               {vectorul cuprinzand "cuvintele"ce formeaza expresia}
                 lung: Integer; {lungimea efectiva a vectorului}
                 a,b: Real; {intervalul de evaluare a functiei}
                 n: Integer {numarul de puncte de evaluare}
               end;
{$I OPERATII.INC}
function Lower(s: String): String;
{ converteste in litere mici si transforma:
    - in 0- ;  (- in (0- ; + in 0+  si  (+ in (0+ }
var t: String; i: Byte;
begin
     t:=''; if s[1] in ['-','+'] then s:='0'+s;
```

```
      i:=1;
      while i < Length(s) do
            if (Copy(s,i,2)='(-') or  (Copy(s,i,2)='(+') then
               begin Insert('0',s,i+1); Inc(i,2) end
            else Inc(i);
      for i:=1 to Length(s) do
          if s[i] in ['A'..'Z'] then
             t:=t+Chr(Ord(s[i])+Ord('a')-Ord('A'))
          else t:=t+s[i];
      Lower:=t
end;
function EsteIntreg(v:real):boolean;
begin
      EsteIntreg := (v=Int(v)) or (v=0)
end;
```

Pentru a despărți diferitele entități ale unei expresii, sunt necesare să se identifice “tipurile” a câte două caractere vecine din şirul reprezentând expresia algebrică. Dacă `t1` şi `t2` sunt tipurile a două caractere vecine din expresie, atunci, dacă ele sunt diferite, înseamnă că nu fac parte din aceeaşi entitate, prin urmare vor fi cuprinse în două “cuvinte” diferite, motiv pentru care se va insera un spațiu între ele.

```
function TipCaracter(c: Char): Byte;
var tip: Byte;
begin
      case c of
           '0'..'9','.': tip:=1; { cifre }
           'c','o','s','i','n','l','e','x','p',
           'a','b','r','d': tip:=2;
           { cos, sin, ln, exp, abs, rad }
           '(': tip:=3; { paranteze }
           ')': tip:=4;
           '+': tip:=5; { operatii aritmetice }
           '-': tip:=6;
           '*': tip:=7;
           '/': tip:=8;
           '^': tip:=9;
           'X': tip:=10; { variabila }
           'q': tip:=11; { pi }
           '=': tip:=12;
           '#': tip:=13;
           '<': tip:=14;
           '>': tip:=15
      else tip:=100
      end;
      TipCaracter:=tip
end;
function EsteNumar(sir: String): Boolean;
var valoare: Real; cod: Integer;
begin
      Val(sir,valoare,cod); EsteNumar := cod=0
end;
```

Procedura următoare inserează spații în expresia funcției, între caracterele de tipuri diferite. Din `sin(x+3.2)` se va obține `sin ( x + 3.2 )`.

```
procedure PuneSpatiiInExpresie(var E: functie);
{ insereaza spatii in expresia functiei,
  intre caractere de tipuri diferite }
var i,t1,t2: Byte;
begin
  with E do
    begin { se pune X in loc de x, dar nu si la functia exp ! }
      if expresie[1]='x' then expresie[1]:='X';
         if expresie[Length(expresie)]='x' then
```

```
            expresie[Length(expresie)]:='X';
        for i:=1 to Length(expresie) do
            if expresie[i]='x' then
               if (i<Length(expresie)) and (i>1) then
                  if (expresie[i-1]<>'e') and (expresie[i+1]<>'p') then
                     expresie[i]:='X';
        i:=1;
        while (i<Length(expresie)) and (Length(expresie)<256) do
          begin
            t1:=TipCaracter(expresie[i]);t2:=TipCaracter(expresie[i+1]);
            if not ( (t1 in [5,6]) and (t2=1) and
                   (expresie[i-2] in ['=','#','<','>']) ) then
               if (t1<>t2) or ((t1=TipCaracter('(')) and (t2=t1)) or
                  ((t1=TipCaracter(')')) and (t2=t1)) then
                  begin Insert(' ',expresie,i+1); i:=i+1 end;
            i:=i+1
          end
      end
end;
```

Despărțirea în cuvinte se realizează cu procedura următoare:

```
procedure TransformaInVector(var E: functie);
{ desparte in cuvinte pe E.expresie, memorand cuvintele in
E.vector[1..lung] }
var p: Byte; sir: String;
begin
  with E do
       begin
         expresie := expresie+' ) '; sir:=expresie; lung:=0;
         while sir<>'' do
               begin
                 p := Pos(' ',sir); lung := lung+1;
                 vector[lung] := Copy(sir,1,p-1); Delete(sir,1,p)
               end
       end
end;
```

Tabelul priorităților se regăseşte în funcția de mai jos:

```
function Prioritate(c: Char): Byte;
var p: Byte;
begin
  case c of
       '(',')': p:=0;
       '+','-': p:=1;
       '*','/': p:=2;
       '^': p:=3; { ridicarea la putere }
       '=','#','<','>': p:=4; { operatori relationali }
       'c','s','l','e','a','r': p:=5 { cos, sin, ln, exp, abs, rad }
    else p:=5
  end;
  Prioritate := p
end;
```

Evaluarea propriu-zisă a funcției:

```
function ValoareFunctie(E: functie; x: Real): Real;
{ returneaza valoarea functiei E in punctul x }
const max_stiva=100;
var valo,x_1,x_2: Real; i,cod,top1,top2:integer;
  { varfurile celor doua stive }
    Opd: array [1..max_stiva] of Real; { stiva operanzilor }
    Op: array [1..max_stiva] of Char; { stiva operatorilor }
begin { calculul expresiei }
  for i:=1 to max_stiva do
      begin Opd[i]:=0; Op[i]:='@' end;
  top1:=0; top2:=1; Op[top2]:='('; i:=0;
```

```
  with E do
    while (i<=lung) and (top2>0) do
      begin
        Inc(i);
        if EsteNumar(vector[i]) then
           begin
             Inc(top1); Val(vector[i],valo,cod);
             Opd[top1]:=valo
           end
        else
           case vector[i][1] of
             { constanta pi=3.1415926 se da sub forma literei q }
             'q': begin Inc(top1); Opd[top1]:=pi end;
             'X': begin  { variabila x } Inc(top1); Opd[top1]:=x end;
             '(': begin {inceput de bloc} Inc(top2); Op[top2]:='(' end
             else
                begin { operatii binare si unare }
                  while (top2>0) and (not (Op[top2] in ['(',')']))
                        and (Prioritate(Op[top2]) >=
                             Prioritate(vector[i][1])) do
                    begin
                      if top1>1 then x_1:=Opd[top1-1];
                         x_2:=Opd[top1];
                      { functii scrise in OPERATII.PAS }
                      case Op[top2] of
                           '=': valo:=Egal(x_1,x_2);
                           '#': valo:=Diferit(x_1,x_2);
                           '<': valo:=MaiMic(x_1,x_2);
                           '>': valo:=MaiMare(x_1,x_2);
                           '+': valo:=Plus(x_1,x_2);
                           '-': valo:=Minus(x_1,x_2);
                           '*': valo:=Inmultit(x_1,x_2);
                           '/': valo:=Impartit(x_1,x_2);
                           '^': valo:=Putere(x_1,x_2);
                           's': valo:=Sinus(x_2);
                           'c': valo:=Cosinus(x_2);
                           'l': valo:=Logaritm(x_2);
                           'e': valo:=Exponential(x_2);
                           'a': valo:=Modul(x_2);
                           'r': valo:=Radical(x_2);
                       end; { case operator[top2] }
                       if Op[top2] in OperatiiBinare then
                          Dec(top1);
                       if Op[top2] in
                          (OperatiiUnare+OperatiiBinare) then
                          Opd[top1]:=valo;
                       Dec(top2);
                    end; { while }
                  if top2>0 then
                     if (Op[top2]<>'(') or (vector[i]<>')') then
                        begin Inc(top2); Op[top2] := vector[i][1] end
                     else Dec(top2);
                end; { else - operatii }
        end { case vector[i][1] }
     end; { while }
  if top2=0 then ValoareFunctie := Opd[1]
  else ValoareFunctie := infinit
end; {calculul_expresiei}
```

Procedura următoare citeşte date despre funcție: expresia sa, intervalul de tabelare, precum şi numărul de puncte în care se vor calcula valorile funcției, din intervalul precizat.

```
procedure CitesteFunctie(var E: functie);
begin
  with E do
    begin
      WriteLn('Dati expresia !'); Write('f(x)=');
      ReadLn(expresie);
      expresie:=Lower(expresie); WriteLn('Dati intervalul !');
      ReadLn(a,b); Write('Dati nr. de puncte = '); ReadLn(n)
    end;
  PuneSpatiiInExpresie(E); TransformaInVector(E)
end;
```

Următoarea procedură tabelează o funcție `E`:

```
procedure AfiseazaValoriFunctie(var E: functie);
{ E este transmis prin referinta, pentru economie de memorie }
var i: Integer; x,y: Real;
begin
  for i:=0 to E.n-1 do
      begin
        x:=E.a+i*(E.b-E.a)/(E.n-1); y:=ValoareFunctie(E,x);
        if DifInf(y) then WriteLn(i+1:3,'. ',x:10:7,' -> ',y:10:7)
        else WriteLn(i+1:3,'. ',x:10:7,' -> nedefinit !');
        if (i+1) mod 24 = 0 then ReadKey
      end;
  ReadKey
end;
var E: functie;
begin
  CitesteFunctie(E);
  AfiseazaValoriFunctie(E)
end.
```

6. Exerciții recapitulative

1. Fie funcția :

```
function Invers(S: String): String;
var t: String; i: Integer;
begin
     t:='';
     for i:= Length(s) downto 1 do t:=t+s[i];
     Invers:=t
end;
```

Scrieți o funcție: `function EstePalindrom(S: String): Boolean;`
astfel incât `EstePalindrom('AERISIREA') = True`,
iar `EstePalindrom('ABRACADABRA') = False`.

2. Rescrieți subprogramele de lucru cu şiruri de caractere:

3. Să se definească un tip de date pentru numere naturale foarte mari şi să se scrie proceduri care să adune şi să scadă astfel de numere.

4. Elaborați o funcție care să determine diferența între două momente de timp date prin oră, minute şi secunde.

5. Scrieți un program care să tabeleze o funcție, definită în textul programului, adică să afişeze valorile funcției în `n` puncte dintr-un interval dat `[a, b]`.

6. Îmbunătățiți programul `AdmitereLiceu` (din secțiunea 3 a acestei lecții) pentru a avea posibilitatea de a şterge, insera un elev sau modifica datele despre un elev. De asemenea, programul să permită listarea tuturor elevilor care au mediile cuprinse între două valori date.

7. Este consistentă următoarea definiție recursivă:

```
function Test(x: Integer);
begin
   if x=0 then Test:=1
   else if x=1 then Test:=Test(x+1) else Test:=Test(x+2)+x
end;
```

8. Ce afişează programul de mai jos:

```
program Test;
var x: Integer;
function F(x: Integer): Integer;
begin x:=x+1; F:=x*x end;
begin x:=2; WriteLn(x,':',F(x),':',x) end.
```

Dar dacă `F` ar avea antetul `function F(var x: Integer): Integer` ?

9. Ce erori sintactice sunt în programul de mai jos ?

```
program Test;
function F(var x: Integer): Integer;
begin x:=x+1; F:=x end;
begin
  WriteLn(2,':',F(2))
end.
```

10. Rezolvă problema mutării a `n` discuri de pe turnul `p` pe turnul `q`, în problema Turnurilor din Hanoi, următoarea procedură? Este ea o procedură definită consistent ?

```
procedure Hanoi2(n,p,q: Integer);
var r: Integer;
begin
  r:=6-p-q;
  if n=1 then WriteLn(p,'-',q)
  else
    begin
      WriteLn(p,'-',r); Hanoi2(n-1,p,r); WriteLn(r,'-',q)
    end
end.
```

Semnificația procedurii este: pentru a muta `n` discuri de la turnul `p` la turnul `q` se mută discul de deasupra de pe turnul `p` pe turnul `r`, apoi se mută, recursiv, cele `n-1` discuri de pe turnul `p` pe turnul auxiliar `r`, după care se aduce discul de pe `q` pe `r`.

11. Scrieți o procedură iterativă care să calculeze, cu ajutorul unei stive, suma `S=1+2+3+...+n`, respectiv produsul `P=1×3×5×...×(2n-1)`.

12. Funcționează programul următor? Dacă da, ce va afişa?

```
program Test;
var y,x: Integer;
function F(var z,x: Integer): Integer;
begin z:=x-y; x:=x+1; F:=x+y+z end;
begin y:=2; x:=y+5; WriteLn(x,':',y,':',F(x,y),':',x,':',y); end.
```

13. Scrieți un program care să afişeze literele unei propoziții la întâmplare pe ecran, apoi să le deplaseze către poziția lor finală, pentru ca şirul să fie scris pe linia din mijloc a ecranului.

14. Ce va afişa programul următor? Ar fi posibil un antet cu `var` pentru funcția `G` ?

```
program Test;
function F(var x: Integer): Integer;
begin x:=x+1; F:=x end;
function G(x,y: Integer): Integer;
begin G:=x+y end;
var y: Integer;
begin y:=2; WriteLn(G(F(y),F(y))) end.
```

15. Un triunghi este definit prin coordonatele vârfurilor sale. Scrieți funcții care, pentru două triunghiuri date, să studieze dacă: a) au aceeaşi arie; b) sunt asemenea; c) primul este în interiorul celui de al doilea.
16. Fie o tablă de şah de dimensiune `n×n`, pe care sunt plasate două dame, prima pe linia `l1` şi coloana `c1`, iar a doua pe linia `l2` şi coloana `c2`. Scrieți o funcție booleană care să verifice dacă damele se atacă între ele sau nu.
17. Scrieți o procedură `Divide(a,b: Integer; c,d: Integer)` care să împartă, prin scăderi repetate, pe `a` la `b`, obținându-se câtul `c` şi restul `d`.
18. Scrieți o procedură care pune într-un vector, în primele `n` componente, primele `n` numere prime care, citite invers, sunt tot numere prime.

Lecția 6
METODE DE ELABORARE A ALGORITMILOR

1. Generarea de mulțimi

Generarea permutărilor

În unele probleme de enigmistică, numite *anagrame*, se cere să se aşeze literele unui cuvânt în toate modurile posibile. Astfel, de pildă, pentru cuvântul "*opt*", se vor obține cuvintele: *opt, otp, pot, pto, top, tpo*.

O aranjare oarecare a tuturor literelor unui cuvânt se numeşte **permutare**. De fapt, putem să numerotăm literele (sau orice gen de obiecte care trebuie permutate), astfel încât, dacă avem n litere (obiecte), o permutare va însemna un vector $p=(p_1,p_2,...,p_n)$, astfel încât mulțimea acestor elemente să fie tot una cu mulțimea $\{1,2,...,n\}$.

Astfel, de pildă, pentru $n=3$ spunem că permutările de gradul 3 sunt:

$(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$, $(3,2,1)$.

Observăm că ordinea în care am scris permutările nu este oarecare; dacă le privim ca numere în baza $n+1$, atunci ele sunt scrise în ordine crescătoare, numită şi *ordine lexicografică* (vezi permutările cuvântului *opt*!).

Referitor la permutările de un anumit grad, se pun următoarele două probleme: *câte sunt* şi *care sunt* ele.

Numărul permutărilor

Pentru a rezolva această problemă, să observăm că numărul permutărilor de grad 1 este 1, a celor de grad 2 este $2=1\times2$, a celor de grad 3 este $6=1\times2\times3$ şi să facem ipoteza că numărul permutărilor de grad n va fi $1\times2\times...\times n$, adică $n!$. Pentru a demonstra că aşa este, vom considera că avem generate permutările de grad $n-1$. Pentru a le obține pe cele de grad n va trebui să punem numărul n înaintea fiecărui element component al fiecărei permutări de grad $n-1$, precum şi la sfârşitul acestora. Prin urmare, dacă avem $(n-1)!$ permutări de gradul $n-1$, vom obține din ele $(n-1)!\times(n-1+1)=(n-1)!\times n=n!$ permutări de grad n, ceea ce ne arată că ipoteza noastră se adevereşte. Procedeul folosit de noi în demonstrarea acestei formule este cunoscut sub denumirea de *inducție matematică*.

Generarea lexicografică a permutărilor

Există mai multe procedee de obținere a permutărilor, unul din ele fiind prezentat şi în demonstrația anterioară. În continuare vom prezenta un algoritm foarte cunoscut, care generează permutările în ordine lexicografică:

- plecăm de la permutarea identică: $p=(1,2,3,...,n)$, cea mai mică din punct de vedere lexicografic;
- să presupunem că avem construită permutarea $p=(p_1,p_2,...,p_n)$. Pentru a obține permutarea imediat următoare, vom determina cel mai mare indice i astfel încât $p_i<p_{i+1}$. Elementul p_i va trebui să fie înlocuit cu cel mai mic dintre elementele de după p_i, mai mare ca p_i; fie acesta p_k. După ce se schimbă p_i cu p_k, se inversează ultimele $n-i$ elemente, pentru a apărea în ordine crescătoare.

Fie `type vector=array[1..10] of Integer`. Putem construi toate permutările de grad n apelând procedura `NextPerm` în felul următor:

```
gata:=True; repeat NextPerm(n,p); if not gata then Scrie(p,n) until gata;
```

Aici, `gata` este o variabilă logică indicând terminarea determinării permutărilor, iar `Scrie` este o procedură ce va afişa primele n componente din vectorul p. Procedura `NextPerm` este descrisă mai jos:

```
procedure NextPerm(n: Integer; var p: vector);
var i,j,k: Integer;
   procedure Schimba(var x,y: Integer);
   var aux: Integer; begin aux:=x; x:=y; y:=aux end;
begin
   if gata then  begin  for i:=1 to n do p[i]:=i; gata:=False end
   else begin
        i:=n-1; while (i>0) and (p[i]>p[i+1]) do i:=i-1;
        if i=0 then gata:=True
        else begin
               k:=n; while p[i]>p[k] do k:=k-1;
               Schimba(p[i],p[k]);
               for j:=1 to (n-i) div 2 do Schimba(p[i+j],p[n-j+1])
             end
         end
end;
```

Generarea combinărilor

O firmă doreşte să trimită m oameni într-o delegație. Firma are n angajați. Se pune problema determinării tuturor modurilor în care se poate forma delegația. Această problemă se numeşte a generării **combinărilor de n elemente luate câte m**. De pildă, pentru m=2 şi n=3, considerând, ca şi în cazul permutărilor, că avem mulțimea {1,2,...,n} acestea sunt: (1,2),(1,3),(2,3).

Pentru a genera în ordine lexicografică vectorii care corespund combinărilor de m elemente luate câte n, vom pleca de la "cel mai mic" vector: $C=(1,2,...,m)$. Ultimul vector, "cel mai mare", va fi $C=(n-m+1,n-m+2,...,n-m+m)$, astfel încât, presupunând că avem construit vectorul $C=(c_1,...,c_m)$, pentru a-i determina succesorul, vom găsi cel mai mare indice i pentru care relația $c_i=n-m+i$ nu mai este îndeplinită, deci avem $c_i<n-m+i$. Vectorul succesor va fi vectorul $(c_1,...,c_i-1,c_i+1,c_i+2,...,c_i+n-i+1)$.

```
procedure NextComb(m,n: Integer; var C: vector);
var i,j: Integer;
begin
  if gata then begin for i:=1 to m do C[i]:=i; gata:=False end
  else begin
         i:=m; while (i>=1) and (C[i]=n-m+i) do i:=i-1;
         if i=0 then gata:=True
         else begin C[i]:=C[i]+1; for j:=i+1 to m do C[j]:=C[j-1]+1 end
       end
end;
```

Generarea tuturor combinărilor se va face prin:

```
gata:=True; repeat NextComb(m,n,C); if not gata then Scrie(C,m) until gata
```

Generarea aranjamentelor

Presupunând că într-o echipă de m oameni aleşi din n salariați ai unei firme, trebuie să existe o ierarhie bine stabilită (o relație de ordine totală), atunci va trebui ca să permutăm în toate felurile posibile combinările de m oameni luați câte n. Mulțimile ce se obțin se numesc **aranjamente de n elemente luate câte m**.

În continuare prezentăm un program demonstrativ pentru a obține aranjamentele de n elemente luate câte m. Propunem cititorului să folosească aceste proceduri ori de câte ori va avea nevoie să genereze astfel de mulțimi (de pildă în unele jocuri de logică).

```
program Generari;
uses Crt;
const max=10;
type vector = array[1..max] of Integer;
var gata_p, gata_c: Boolean; m,n,i: Integer; P,C,A: vector;
procedure Scrie(x: vector; n: Integer);
var i: Integer;
```

```
begin
  for i:=1 to n do Write(x[i],','); ReadLn
end;
procedure NextPerm(n: Integer; var P: vector);
var i,j,k: Integer;
  procedure Schimba(var x,y: Integer);
  var aux: Integer; begin aux:=x; x:=y; y:=aux end;
begin
  if gata_p then
     begin
       for i:=1 to n do P[i]:=i;
       gata_p:=False
     end
  else begin
         i:=n-1;
         while (i>0) and P[i]>P[i+1]) do i:=i-1;
         if i=0 then gata_p:=True
         else begin
                k:=n; while P[i]>P[k] do k:=k-1;
                Schimba(P[i],P[k]);
                for j:=1 to (n-i) div 2 do Schimba(P[i+j],P[n-j+1])
              end
       end
end;
procedure NextComb(m,n: Integer; var C: vector);
var i,j: Integer;
begin
  if gata_c then begin for i:=1 to m do C[i]:=i; gata_c:=False end
  else begin
         i:=m;
         while (i>=1) and (C[i]=n-m+i) do i:=i-1;
         if i=0 then gata_c:=True
         else begin
                C[i]:=C[i]+1;
                for j:=i+1 to m do C[j]:=C[j-1]+1
              end
       end
end;
```

```
begin { generarea aranjamentelor }
     ClrScr; WriteLn('Generari de m-aranjamente luate cite n');
     Write('n='); ReadLn(n); Write('m='); ReadLn(m);
     gata_p:=True; gata_c:=True;
     repeat
        NextComb(m,n,C);
        if not gata_c then
           begin
             gata_p:=True;
             repeat
               NextPerm(m,P);
               if not gata_p then
                  begin
                    for i:=1 to m do A[i]:=C[P[i]]; Scrie(A,m)
                  end
             until gata_p
           end
     until gata_c; ReadLn
end.
```

Generarea produsului cartezian

Să considerăm că avem o mulțime de `m` băieți şi o mulțime de `n` fete. Ne punem problema de a cupla băieții cu fetele în toate modurile posibile. Problema nu e grea, două cicluri `for` unul în altul rezolvând-o:

```
program Cuplaj;
var m,n,i,j: Integer;
begin
     WriteLn('Dati m si n'); ReadLn(m,n);
     for i:=1 to m do
         for j:=1 to n do WriteLn('Cuplam baiatul ',i,' cu fata ',j);
     ReadLn
end.
```

De fapt am realizat produsul cartezian al mulțimilor $\{1,2,...,m\}$ şi $\{1,2,...,n\}$. Problema se rezolvă la fel atunci când avem de generat produsul cartezian a p mulțimi, având p cicluri `for` unul într-altul. Însă, când acest număr p este variabil, necunoscut aprioric, problema se rezolvă după cum urmează.

Să presupunem că avem p mulțimi $A_1, A_2, ..., A_m$, unde $A_i=\{1,2,...,n_i\}$(pentru fiecare i de la 1 la p). Să determinăm toate cele $n_1 \times n_2 \times ... \times n_p$ elemente ale produsului cartezian $A_1 \times A_2 \times ... \times A_p$, adică mulțimea tuturor vectorilor v de p elemente care cuplează ordonat elementele din cele p mulțimi. Inițial, vectorul va fi $v=(1,1,...,1)$. Apoi, pentru a determina succesorul (în ordine lexicografică) unui vector v se determină cel mai mare indice i cu $v_i < n_i$, iar vectorul succesor va fi $(v_1,...,v_{i-1},v_i+1,1,...,1)$. Dacă un astfel de indice i nu există, înseamnă că s-a ajuns la ultimul vector $v=(n_1,n_2,...,n_p)$.

Iată, de pildă, un program care ne afişează toate posibilitățile de a cupla pentru dans trei băieți cu patru fete, în două variante de dans:

```
program ProdusCartezian;
type vector=array[1..10] of Integer; vec=array[1..3] of Integer;
var p: Integer; cuplare: vector; gata: Boolean;
const nume: array[1..3,1..4] of String = (('Dan','Mihai','Ion',''),
       ('Anca','Diana','Cristina','Monica'), ('vals','tango','',''));
      n: vec = (3,4,2);

procedure Scrie(v: vector; p: Integer);
var i: Integer;
begin
     for i:=1 to p do Write(nume[i,v[i]],'-'); Write(#8#32); ReadLn
end;
procedure Next(p: Integer; n: vec; var v: vector); { produsul cartezian }
var i: Integer; gasit: Boolean;
begin
     if gata then begin for i:=1 to p do v[i]:=1; gata:=False end
     else
        begin
          i:=p; gasit:=False;
          while (i>0) and (not gasit) do
                if v[i]<n[i] then begin v[i]:=v[i]+1; gasit:=True end
                else begin v[i]:=1; i:=i-1 end;
          if not gasit then gata:=True
        end
end;
begin
     WriteLn(' Generare produs cartezian');
     WriteLn('Cuplare baieti - fete - dans'); WriteLn;
     gata:=True;
     repeat
        Next(3,n,cuplare); if not gata then Scrie(cuplare,3)
     until gata; ReadLn
end.
```

Rezultatul programului este afişarea următoarelor triplete:

```
Generare produs cartezian
Cuplare baieti - fete - dans
Dan-Anca-vals
Dan-Anca-tango
Dan-Diana-vals
```

```
Dan-Diana-tango
Dan-Cristina-vals
Dan-Cristina-tango
Dan-Monica-vals
Dan-Monica-tango
Mihai-Anca-vals
Mihai-Anca-tango
Mihai-Diana-vals
Mihai-Diana-tango
Mihai-Cristina-vals
Mihai-Cristina-tango
Mihai-Monica-vals
Mihai-Monica-tango
Ion-Anca-vals
Ion-Anca-tango
Ion-Diana-vals
Ion-Diana-tango
Ion-Cristina-vals
Ion-Cristina-tango
Ion-Monica-vals
Ion-Monica-tango
```

Generarea submulțimilor unei mulțimi

Să considerăm o mulțime oarecare cu `n` elemente. Ne punem problema determinării tuturor submulțimilor sale, începând cu mulțimea vidă şi terminând cu mulțimea mare însăşi. Vom prezenta o metodă bazată pe generarea produsului cartezian. Astfel, să considerăm **vectorul caracteristic** asociat unei submulțimi a mulțimii inițiale, ca fiind un vector `v` cu `n` elemente, în care `v[i]=1` dacă elementul de pe poziția `i` este luat în submulțimea respectivă, respectiv `v[i]=0` dacă nu. Astfel, vectorul având doar zerouri va corespunde mulțimii vide, iar cel având doar elemente `1` mulțimii întregi. Problema se reduce la a genera toți vectorii cu componente `0` şi `1`, deci a realiza produsul cartezian a `n` mulțimi `{0,1}`.

De fapt, putem spune că acei vectori reprezintă codificarea în baza `2` a numerelor de la `0` la 2^n-1, deci putem să obținem vectorii caracteristici prin transformarea (şi memorarea rezultatului) în baza `2` a numerelor de la `1` la 2^n-1. Lăsam cititorului drept exercițiu rezolvarea acestei probleme, pe una din căi.

2. Metode simple de sortare

Problema generală este să aşezăm în ordine crescătoare elementele unui vector `x:array[1..max]of...`, de dimensiune `n`. În această secțiunea a lecției vom da unele metode simple şi mai cunoscute de sortare. Ele nu sunt dintre cele mai eficiente, dar asupra metodelor rapide vom reveni în secțiunea referitoare la tehnica "*divide et impera*".

Metoda selecției directe şi metoda "*bubble-sort*"

Aceste metode au fost deja explicate, în lecția 4:

- **Sortarea prin selecție directă:**

```
pentru i:=1 la n-1 execută
       pentru j:=i+1 la n execută
              dacă x[j]<x[i] atunci schimbă(x[i], x[j]).
```

- **Sortarea prin metoda "bulelor" (*bubble-sort*)**

```
repetă
   gata:=adevărat;
   pentru i:=1 la n-1 execută
        dacă x[i+1]<x[i] atunci
             început
               schimbă(x[i], x[i+1]); gata:=fals
             sfârsit
până când gata.
```

Sortarea prin numărare

Se numără, pentru fiecare element a_i din vectorul de sortat a câte elemente mai mici ca el există. Aceste numere sunt memorate într-un vector K. Pe baza acestui vector se rearanjează elementele vectorului a pentru a forma un vector b, care se va copia apoi în a. Presupunem că elementele din vectorul a sunt distincte.

```
procedure NumSort(n: Integer; var A: vector);
var B,K: vector; i,j: Integer;
begin
   for i:=1 to n do K[i]:=0;
   for i:=1 to n-1 do
       for j:=i+1 to n do
           if A[i]<A[j] then K[j]:=K[j]+1
           else K[i]:=K[i]+1;
   for i:=1 to n do B[K[i]+1]:=A[i];
       for i:=1 to n do A[i]:=B[i]
end;
```

Sortarea prin inserție

Metodele de inserție se bazează pe considerarea, pe rând, a fiecărui element al vectorului şi inserarea acestuia în subşirul ordonat crescător, anterior creat, din elementele precedente.

- **Sortarea prin inserție directă** constă în inserarea în subşirul $a_1<a_2<...<a_{j-1}$ a elementului a_j. Astfel, vom compara pe a_j cu a_{j-1}, a_{j-2}, ..., până ajungem la a_i cu $a_i<a_j$. Deci vom insera pe a_j după a_i, fapt ce implică deplasarea spre dreapta a unei secvențe.

Următoarea procedură combină inserarea cu deplasarea spre dreapta.

```
procedure InsDirSort(n: Integer; var A: vector);
var aux: Integer; i,j,k: Integer; gata: Boolean;
begin
  for i:=2 to n do
      begin
        aux:=A[i]; k:=0; j:=i-1; gata:=False;
        while (j>=1) and (not gata) do
          if A[j]>aux then
                 begin A[j+1]:=A[j]; j:=j-1 end
          else begin k:=j; gata:=True end;
        A[k+1]:=aux
      end
end;
```

- **Sortarea prin metoda "*shell*"** ("decojire"). Prin *h-sortare* (unde $h>0$, $h\in N$, numit *increment*) vom înțelege sortarea (prin selecție directă sau "*bubble-sort*") a următoarelor secvențe, care formează o partiție a mulțimii $\{a_1,...,a_n\}$:

$$\{a_1, a_{1+h}, a_{1+2h}, ...\}; \{a_2, a_{2+h}, a_{2+2h}, ...\}; ...; \{a_h, a_{2h}, ...\}.$$

O h-sortare permite elementelor să se deplaseze spre stânga cu h poziții, iar după o h-sortare, orice element se află la o distanță egală cu cel mult $h-1$ poziții de poziția sa finală. Metoda "*shell*" constă în alegerea unei secvențe de k incremenți $h_1>h_2>...>h_k=1$ şi de a face o h_1-sortare, apoi o h_2-sortare, până la o 1-sortare. Modul de alegere a acestor incremenți reglează eficiența metodei "*shell*". În procedura ce urmează se pleacă cu $h_1=n$, apoi, în fiecare etapă, incrementul se înjumătățeşte.

```
procedure ShellSort(n: Integer; var A: vector);
var i,h: Integer; gata: Boolean;
begin
  h:=n;
  while h>1 do
```

```
          begin
            h:=h div 2;
            repeat
             gata:=True;
             for i:=1 to n-h do
                 if A[i]>A[i+h] then
                    begin Schimba(A[i],A[i+h]); gata:=False end
            until gata
          end
      end;
```

3. Metoda "*greedy*"

Prezentare generală

Această metodă se foloseşte în problemele în care, dată fiind o mulțime `A` cu `n` elemente, se cere să se determine o submulțime `B` a lui `A`, care să îndeplinească anumite condiții (eventual un anumit criteriu de optim).

Metoda "*greedy*" este următoarea:

- se inițializează mulțimea `B` la mulțimea vidă;
- se alege un anumit element din `A`;
- se verifică dacă elementul ales poate fi adăugat mulțimii `B`;
- procedeul continuă aşa, repetitiv, pînă ce au fost determinate toate elementele din `B`.

Despre această tehnică de programare, foarte utilizată dealtfel, s-a discutat în lecția 4, la vectori, unde anterior fuseseră prezentate suficiente exemple. Există şi o serie de probleme celebre, în a căror rezolvare se poate folosi această strategie: *problema continuă a rucsacului; algoritmul lui Dijkstra pentru drumuri minime*.

Problema continuă a rucsacului

Cu ajutorul unui rucsac de greutate maximă admisibilă `GG` trebuie să se transporte o serie de obiecte din `n` disponibile, având greutățile G_1, G_2, ..., G_n, aceste obiecte fiind de utilitățile C_1, C_2, ..., C_n. Dacă pentru orice obiect `i` putem să luăm doar o parte $x_i \in [0,1]$ din el, atunci spunem că avem **problema continuă a rucsacului**, pe care o vom prezenta aici, iar dacă obiectul poate fi luat doar în întregime sau nu, spunem că avem **problema discretă a rucsacului**, pe care o vom prezenta în secțiunea următoare.

În problema continuă a rucsacului, prin raportarea utilităților la greutăți obținem utilitățile pe unitate de greutate, astfel încât va trebui să ordonăm obiectele în funcție de aceste raporturi şi să le încărcăm, pe cât posibil, în întregime în rucsac, până când acesta se umple. Astfel, reprezentând soluția în vectorul `x`, vom pune la început `x[i]:=1`, până când greutatea rămasă disponibilă, notată `GGr`, nu mai permite punerea unui obiect în întregime. Atunci, vom face `x[i]:=GGr/G[i]`, ultimul obiect fiind "tăiat".

Programul de mai jos rezolvă această problemă; mai întâi ordonează obiectele, apoi le afişează ordonate, după care aplică metoda descrisă, la fiecare pas actualizând greutatea rămasă disponibilă `GGr`, după formula: `GGr:=GG-G[i]`.

```
program Rucsac;
const max=5;
var C,G,X: array[1..max] of Real;
  n,i,j:Integer;GG,GGr,aux:Real;
begin
  Write('Nr. obiecte = ');
  ReadLn(n);
  for i:=1 to n do
      begin
        Write('C[',i,']=');
        ReadLn(C[i]);
```

```
        Write('G[',i,']=');
        ReadLn(G[i])
      end;
  Write('Greut. max. = ');
  ReadLn(GG);
  for i:=1 to n-1 do
    for j:=i+1 to n do
     if C[j]/G[j]>C[i]/G[i] then
        begin
          aux:=C[j]; C[j]:=C[i];
          C[i]:=aux; aux:=G[j];
          G[j]:=G[i]; G[i]:=aux
        end;
  WriteLn('Am ordonat ...');
 for i:=1 to n do
     WriteLn('C[',i,']=',C[i]:5:2,
' G[',i,']=',G[i]:5:2,
' -> ',C[i]/G[i]:5:2);
 GGr:=GG; i:=1;
 while (i<=n) do
  if GGr > G[i] then
     begin
       X[i]:=1;
       GGr:=GGr-G[i]; i:=i+1
     end
  else
    begin
      X[i]:=GGr/G[i];
      for j:=i+1 to n do X[j]:=0;
      i:=n+1
    end;
 for i:=1 to n do
   WriteLn('X[',i,']=',X[i]:5:2);
 ReadLn
end.
```

4. Metoda "*backtracking*"

Prezentare generală

Există multe probleme a căror soluție se poate reprezenta sub forma unui vector $x=(x_1, ..., x_n) \in S$, în care $S=S_1 \times S_2 \times ... \times S_n$, unde $S_1,...,S_n$ sunt mulțimi finite, cu $|S_i|=s_i$ elemente. (De pildă, să considerăm problema aşezării a n dame pe o tablă de şah $n \times n$. În acest caz, fireşte vom avea exact o damă pe fiecare coloană. Deci dama k va sta pe coloana k şi pe una din cele n linii, notată $x[k]$. Aşadar, o soluție a acestei probleme poate fi reprezentată doar prin vectorul liniilor pe care sunt aşezate cele n dame, coloanele subînțelegându-se.) Pentru fiecare astfel de problemă sunt date anumite relații între componentele x_1, ..., x_n ale vectorului x, numite *condiții interne*. S se numeşte *spațiul soluțiilor posibile*. Soluțiile posibile, care satisfac condițiile interne se numesc *soluții rezultat*. (În cazul problemei damelor, se cere ca damele să nu se atace. Aşezându-le câte una pe fiecare coloană, ele nu se vor ataca vertical. Pentru a nu se ataca orizontal, trebuie ca să nu existe două dame pe aceeaşi linie (deci două dame i şi j cu $x[i]=x[j]$. De asemenea, trebuie ca damele să nu se atace diagonal.). Astfel de probleme pot fi soluționate folosind **metoda "*backtracking*"** (în caz că nu se cunoaşte o metodă mai bună).

Metoda "*back-tracking*" evită generarea tuturor soluțiilor posibile. Elementele vectorului x primesc, pe rând, valori. Astfel, lui x_k i se atribuie o valoare numai dacă au fost deja atribuite valori lui x_1, ..., x_{k-1}. Mai mult, odată o valoare pentru x_k stabilită, nu se trece direct la atribuirea de valori lui x_{k+1}, ci se verifică nişte *condiții de continuare* referitoare la x_1, ..., x_k. Aceste condiții stabilesc situațiile în care are sens să trecem la determinarea lui x_{k+1}. Neîndeplinirea lor exprimă faptul că, oricum am alege x_{k+1}, ..., x_n, nu vom putea ajunge la o soluție rezultat, adică la o

soluție pentru care condițiile interne să fie satisfăcute. Dacă condițiile de continuare nu sunt îndeplinite, se va alege alt x_k din S_k, iar dacă S_k s-a epuizat ne întoarcem la S_k.

Putem vedea vectorul x ca pe o stivă asupra căreia se fac o serie de operații de intrare şi ieşire, atât timp cât stiva nu este vidă. Prin urmare, metoda va admite şi o variantă recursivă.

- Presupunând stabilite condițiile de continuare (sub forma unei funcții depinzând de valorile stabilite până acum) $\varphi_k(x_1, ..., x_k)$ pentru $x_1, ..., x_k$, avem:

```
type vector = array[1..max] of  TIP; var α: TIP;
procedure BackTracking(n: Integer; var x: vector);
var  k: Integer; cont: Boolean;
begin
  k := 1;
  while k > 0 do begin
    cont := False;
    while (mai există o valoare α∈Sk netestată) and (not cont) do
      begin
        xk := α;
       if φk(x1, ..., xk) then cont := True
      end;
    if not cont then k := k - 1
    else if k = n then Scrie(x) else k := k + 1
  end
end;
```

- În general, însă, x_k ia valori numere întregi între 1 şi n. Astfel, putem rescrie procedura BackTracking anterioară sub forma de mai jos, în care cele n valori sunt date lui x[k]în felul următor: mai întâi fiecare x[k] primeşte valoarea 0, apoi, la fiecare încercare de a da o nouă valoare, se face incrementarea x[k]:=x[k]+1.

```
type vector = array[1..max] of TIP;
procedure BackTracking(n: Integer; var x: vector);
var k: Integer; cont: Boolean;
begin
  k := 1; x[k]:=0;
  while k > 0 do
        begin
          cont := False;
          while (x[k] < n) and (not cont) do
                begin
                  x[k]:=x[k]+1
                  if PotContinua(x,k) then
                     cont := True
                end;
           if not cont then
              k := k - 1
           else if k = n then Scrie(x)
                else begin k := k + 1; x[k]:=0 end
        end
end;
```

- Procedura admite şi o *variantă recursivă* (se apelează BackTrackRec(1,n,x)). Astfel, alegerea unei valori pentru x[i] va determina şi un apel recursiv al acestei proceduri pentru i+1.

```
procedure BackTrackRec(i,n: Integer; var x: vector);
var k: Integer;
begin
  k := 1;
 while k <=n do
       begin
          x[i]:=k;
```

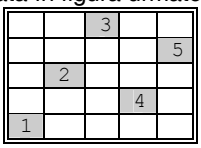
```
          if PotContinua(x,k) then
             if k = n then Scrie(x)
             else BackTrackRec(i+1,n,x);
          k := k+1
        end
end;
```

Mersul înapoi se realizează după ce `k` primeşte ultima valoare, iar `PotContinua` returnează `False`.

Printre exemplele celebre de probleme în a căror rezolvare se poate folosi această metodă de programare, amintim: *problema damelor; problema labirintului; colorarea vârfurilor unui graf (a unei hă]i); generarea permutărilor; problema circuitului hamiltonian*.
Între condițiile interne şi cele de continuare există o strânsă legătură, alegerea cât mai bună a condițiilor de continuare reducând calculele. Rezolvitorului îi revine problema soluționării, astfel, a două probleme importante legate de aplicarea metodei "*back-tracking*": memorarea soluției şi scrierea funcției de continuare.

Problema damelor

Problema constă în a aşeza `n` dame pe o tablă de şah `n×n`, astfel încât damele să nu se atace. O soluție pentru `n=5` este dată în figura următoare.



Vom aşeza, pe rând, câte o damă pe fiecare coloană `k`, dama de pe coloana `k` fiind aşezată pe linia `x[k]`. Astfel, damele nu se vor ataca pe verticală Mai rămâne ca ele să nu se atace pe orizontală şi pe diagonală. Este de ajuns ca dama `k` să nu se atace, astfel, cu nici una din damele dinaintea sa, deci cu damele `i`, cu `i=1,...,k-1`. Condițiile de continuare sunt, deci:

`x[i]≠x[k], ∀i=1,...,k-1` (neatac pe linii (orizontală)) şi
`|x[i]-x[k]|≠k-i, ∀i=1,...,k-1` (neatac pe diagonală).

Acestea fiind spuse, putem scrie funcția de continuare astfel:

```
function PotContinua(x: vector; k: integer): Boolean;
var atac: Boolean; i: Integer;
begin
  atac := false; i := 1;
  while (i < k) and (not atac) do
        if (x[i] = x[k]) or (abs(x[i]-x[k]) = k-i) then
           atac := True
        else i := i+1;
  PotContinua := not atac
end;
```

1.Varianta iterativă:

```
program ProblemaDamelor;
uses Crt;
const max=10;
type vector = array[1..max] of Integer;
var NrSol: Integer;
procedure Scrie(n: Integer; x: vector);
var i: Integer;
begin
  Inc(NrSol);
  WriteLn('Solutia nr. ',NrSol);
  for i := 1 to n do
      WriteLn('Dama de pe coloana ',i,' e pe linia ',x[i]);
  WriteLn; ReadLn
end;
```

```
function PotContinua(x: vector; k: integer): Boolean;
{ ..... }
procedure Dame(n: Integer; var x: vector);
var k: Integer; cont: Boolean;
begin
  k := 1; x[k] := 0;
  while k > 0 do
    begin
      cont := False;
      while (x[k] < n) and (not cont) do
         begin
            x[k] := x[k] + 1;
            if PotContinua(x,k) then cont := True
         end;
      if not cont then k := k - 1
      else
        if k = n then Scrie(n,x) else begin k := k + 1; x[k] := 0 end
   end
end;
var AsezareDame: vector; NrDame: integer;
begin
     ClrScr; WriteLn(' Problema damelor '); WriteLn;
     WriteLn; NrSol:=0; NrDame := 5; Dame(NrDame,AsezareDame)
end.
```

2. Varianta recursivă

Varianta recursivă se obține din programul anterior înlocuind procedura `Dame` cu procedura `Dama`, de mai jos. De asemenea, apelul din blocul principal devine: `Dama(1,NrDame,AsezareDame).`

```
procedure Dama(i,n: Integer; var x: vector);
var k: Integer; cont: Boolean;
begin
  k := 1;
  while k <= n do
   begin
     x[i] := k;
     if PotContinua(x,i) then if i=n then Scrie(n,x) else Dama(i+1,n,x);
     k := k + 1
   end
end;
```

Problema discretă a rucsacului

Această problemă, numită şi "*problema 0/1 a rucsacului*", este similară celei prezentate la metoda "*greedy*", deosebirea fiind că aici obiectele care se pun în rucsac nu pot fi tăiate. Problema se poate soluționa şi pe alte căi mai eficiente, însă vom prezenta rezolvarea sa prin metoda "*back-tracking*", din considerente de ordin didactic. Astfel, problema în cauză se reduce la a genera toți vectorii $x=(x_1,x_2,...,x_n)$, cu $x_i \in \{0,1\}$. Fiecare vector reprezintă o modalitate de a umple rucsacul ($x_i=1 \Leftrightarrow$ obiectul `i` s-ar lua în rucsac), deci va avea un câştig curent asociat (o utilitate) (`CC`) şi o greutate curentă (`Greut`), care nu trebuie să depăşească valoarea maximă admisibilă: `GG`. Dintre toți aceşti vectori - care ar putea fi generați şi pe alte căi, de exemplu ca elementele unui produs cartezian, sau prin transformări în baza 2 a primelor `2n-1` numere - se va alege vectorul care are utilitatea maximă printre toți vectorii. Acest vector - cel mai bun - se va copia într-un vector de acelaşi fel, numit în program `Iau`, cu convenția că `Iau[i]=1` dacă obiectul `i` se pune în rucsac, respectiv `Iau[i]=0` dacă nu.

```
program Rucsac01;
const max=10;
var CMax,CC,GG:Integer; C,G,X,Iau: array[1..max] of Integer;
    { CMax,CC = cistig maxim, curent, GG = greutatea maxima posibila }
    n,k,i: Integer; cont: Boolean;
function PotContinua(k: Integer): Boolean;
{ pot continua daca nu se depaseste GG }
```

```
var i: 1..max; Greut: Integer;
begin
  Greut:=0;
  for i:=1 to k do if X[i]=1 then Greut:=Greut+G[i];
  PotContinua:=Greut<=GG
end;
procedure BackTrack;
begin
  k:=1; X[k]:=-1; CMax:=0;
  while k>0 do
    begin
      cont:=False;
      while (X[k]<1) and (not cont) do
        begin X[k]:=X[k]+1; cont:=PotContinua(k) end;
      if cont then
         if k=n then
            begin
              CC:=0;
              for i:=1 to n do if X[i]=1 then CC:=CC+C[i];
              if CC>=CMax then
                 begin CMax:=CC; for i:=1 to n do Iau[i]:=X[i] end
            end
         else begin k:=k+1; X[k]:=-1 end
      else k:=k-1
    end
end;
begin
  Write('n='); ReadLn(n);
  for i:=1 to n do
      begin
        Write('C[',i,']='); ReadLn(C[i]); Write('G[',i,']='); ReadLn(G[i])
      end;
  Write('GG='); ReadLn(GG); BackTrack;
  WriteLn('O sol. cu cistig maxim:');
  for i:=1 to n do if  Iau[i]=1 then  WriteLn('Se ia obiectul ',i);
  WriteLn('Câstig = ',Cmax); ReadLn
end.
```

Problema labirintului

Există şi situații când soluțiile unor probleme, rezolvabile prin metoda "*back-tracking*" se pot da sub forma unei matrice, care să conțină, în nişte "căsuțe", numerele `1`,`2`,...,`p`, unde `p` este lungimea vectorului asociat soluției (în varianta de până acum); `p` poate varia de la caz la caz. Un bun exemplu îl constituie următoare problemă.

Se dă un labirint memorat sub forma unei matrice `Labirint` de elemente `0` şi `1`, în care unitățile corespund spațiilor pe unde se poate trece, iar zerourile zidurilor. Un şoricel pus într-o anumită căsuță a labirintului ((`i_initial`,`j_initial`)) va trebui să ajungă într-o altă casuță a labirintului, unde se află o bucățică de caşcaval ((`i_final`,`j_final`)). El se poate mişca doar ortogonal, nu şi diagonal. Pentru a determina toate posibilitățile de a ajunge la caşcaval, fără a trece de mai multe ori prin acelaşi loc, vom folosi metoda "*back-tracking*", într-o variantă recursivă, cu unele modificări. Astfel, nu vom memora drumurile parcurse de şoricel sub forma unui vector `x`, deoarece nu ştim cât de mare poate ajunge acest vector la un moment dat. Vom folosi, în schimb, o matrice `Traseu` asociată tablei. Convenim ca `Traseu[i,j]` să fie o valoare `pas` dacă şoricelul trece la pasul `pas` pe acolo, în drumul său, respectiv `0`, în orice alt caz. De asemenea, observăm că şoricelul, dacă se află în poziția (`i,j`), nu se poate deplasa decât în patru direcții, sus, jos, stânga şi dreapta. Astfel, din pozi]ia (`i,j`), unde a ajuns la pasul `pas`, va trece într-o poziție nouă (`i_nou`,`j_nou`), la pasul `pas+1`. Noile coordonate se obțin din cele vechi prin adăugarea valorilor `-1`, `0`, sau `1`, în conformitate cu cele patru direcții. Acest lucru se va realiza folosind două şiruri speciale de numere, notate prin `oriz` şi `vert`. Programul de mai

jos foloseşte o matrice constantă pentru un labirint de dimensiune 8×10. Una din soluțiile programului pentru datale de intrare (4,4) şi (1,6) este dată de matricea de mai jos:

0	0	0	0	0	6	0	0	0	0
0	0	0	0	0	5	0	0	0	0
0	0	0	2	3	4	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

```
program ProblemaLabirintului;
const m=8; n=10;
type sir=array[1..4] of ShortInt; matrice=array[1..m,1..n] of Byte;
const Labirint: matrice=((0,0,0,0,0,1,0,0,0,0), (0,0,0,1,0,1,0,1,0,0,0,0),
  (0,0,0,1,1,1,0,0,0,0),(1,1,1,1,0,1,0,0,0,0),(0,0,0,1,0,1,0,0,0,0),
  (0,1,1,1,1,1,1,1,0,0),(1,1,0,0,1,0,0,0,0,0),(0,0,0,0,1,0,0,0,0,0));
  oriz: sir = (-1,0,1,0);
  vert: sir = (0,1,0,-1);
var Traseu:matrice; i,j,i_initial, j_initial, i_final,j_final: Byte;
procedure Scrie;
var i,j: Integer;
begin
  for i:=1 to m do
      begin for j:=1 to n do Write(Traseu[i,j]:3); WriteLn end;
  WriteLn
end;
procedure Drum(i,j,pas: Byte);
{procedura recursiva de back-tracking}
var i_nou,j_nou: ShortInt; varianta: Byte;
begin
  for varianta:=1 to 4 do
    begin
      i_nou:=i+oriz[varianta]; j_nou:=j+vert[varianta];
      if (i_nou in [1..m]) and (j_nou in [1..n]) then
         if (Labirint[i_nou,j_nou]=1) and (Traseu[i_nou,j_nou]=0) then
            begin
              Traseu[i_nou,j_nou]:=pas;
              if (i_nou=i_final) and (j_nou=j_final) then Scrie
              else Drum(i_nou,j_nou,pas+1);
              Traseu[i_nou,j_nou]:=0
            end
    end
end;
begin
 { se initializeaza matricea Traseu }
 for i:=1 to m do for j:=1 to n do Traseu[i,j]:=0;
 Write('Dati pozitia initiala -> '); ReadLn(i_initial, j_initial);
 Write('Dati pozitia finala   -> '); ReadLn(i_final, j_final);
 Traseu[i_initial, j_initial]:=1; WriteLn('Solutii : ');
 Drum(i_initial, j_initial, 2); ReadLn
end.
```

5. Utilizarea mouse-ului în modul text

Prin **modul text** se înțelege modul de lucru al calculatorului care este prezent imediat după pornirea calculatorului, în sistemul de operare MS-DOS. Acelaşi mod text este utilizat de mediul de programare Turbo-Pascal (deci nu modul grafic).

Ne propunem, în continuare, să realizăm o variantă mult mai atractivă a programului cu labirintul, prezentat în secțiunea anterioară. Astfel, dacă tot vorbim de şoricei, să realizăm un program în care labirintul să se editeze cu ajutorul mouse-ului. De asemenea, vom realiza o animație pentru a arăta traseul şoricelului.
Procedurile de lucru cu mouse-ul în modul text se realizează în limbaj de asamblare, ele utilizând întreruperea de sistem `33h`. Pe cititor nu trebuie să-l intereseze ce se ascunde în spatele lor, ci mai degrabă cum se pot folosi ele. Astfel, avem la dispoziție următoarele proceduri:

- **`procedure MouseInit`** → inițializează mouse-ul;
- **`procedure MouseShow`** → afişează pătrățelul (cursorul) de mouse pe ecran;
- **`procedure MouseHide`** → şterge acest pătrățel;
- **`procedure MouseTData(var buton,x,y:Integer)`** → citeşte date despre mouse:
  - variabila `buton` va conține valoarea butonului de mouse apăsat:
    `0` = nici un buton; `1`= butonul din stânga; `2` = butonul din dreapta;
    `3` = ambele butoane (butonul din mijloc nu este utilizat!).
  - variabilele `x` şi `y` indică poziția curentă a mouse-ului:
    `x` variază între `1` şi `80` → coloana;
    `y` variază între `1` şi `25` → linia.

Acum, pentru a înțelege cum pot fi utilizate aceste proceduri, vom da un exemplu: cursorul de mouse se plimbă pe ecran, în funcție de cum deplasăm noi mouse-ul pe masă. La apăsarea butonului din stânga, în poziția curentă a cursorului va fi afişat un caracter cu codul `ASCII 178`. Procesul se termină la apăsarea butonului din dreapta.

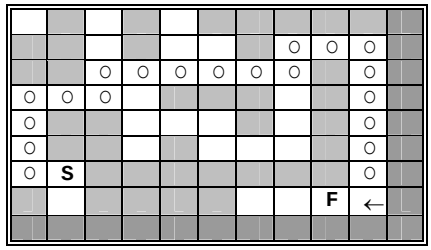
```
program TestMouseText;
uses Crt;
procedure MouseInit; assembler;
asm mov ax,0; int $33 end;
procedure MouseShow; assembler;
asm mov ax,1; int $33 end;
procedure MouseHide; assembler;
asm mov ax,2; int $33 end;
procedure MouseTData(var buton, x, y: Integer);
var bb,xx,yy: Integer;
begin
  asm mov ax,3; int $33; mov bb,bx; mov xx,cx; mov yy,dx end;
  buton:=bb; x:=xx div 8 + 1; y:=yy div 8 + 1
end;
var b,x,y: Integer;
begin
     ClrScr; MouseInit; MouseShow;
     repeat
       MouseTData(b,x,y);
       if b=1 then begin
       {inaintea unei afisari se va ascunde intotdeauna mouse-ul }
            MouseHide; GoToXY(x,y); Write(#178); MouseShow
          end
  until b=2
end.
```

Secțiunea încadrată în programul de mai sus o notăm cu `M` şi ea desemnează toate procedurile de mouse pentru modul text, care vor fi folosite şi în programul următor. Caracterul `#178` este folosit pentru ziduri, care se desenează acționând butonul din stânga. Cu butonul din dreapta vom desemna, pe rând, pozițiile de start şi finiş. Trebuie să avem grijă să nu ieşim din zona încadrată în ecran (de caracterele `#219`). Deplasarea şoriceului va fi realizată folosind caracterele `spațiu` şi `#1` (un joker).

```
program LabirintulMagic;
uses Crt;
M ;
procedure PrintAt(x,y: Integer; sir: String);
begin GoToXY(x,y); Write(sir) end;
function Apartine(x,y,x1,y1,x2,y2: Integer): Boolean;
begin
     Apartine:=(x1<=x) and (x<=x2) and (y1<=y) and (y<=y2)
end;
const m=8; n=10;
type matrice = array[1..m, 1..n] of Byte; sir = array[1..4] of ShortInt;
var Labirint: matrice;
const oriz: sir = (-1,0,1,0); vert: sir = (0,1,0,-1);
var Traseu: matrice; i, j, i_initial, j_initial, i_final, j_final: Byte;
procedure Drum(i,j,pas: Byte);
{procedura recursiva de back-tracking}
var i_nou,j_nou: ShortInt; varianta: Byte;
begin
  for varianta:=1 to 4 do
      begin
        i_nou:=i+oriz[varianta]; j_nou:=j+vert[varianta];
        if (i_nou in [1..m]) and (j_nou in [1..n]) then
           if (Labirint[i_nou,j_nou]=1) and (Traseu[i_nou,j_nou]=0) then
              begin
                Traseu[i_nou,j_nou]:=pas; PrintAt(j_nou,i_nou,#1);
                Delay(100);
                if (i_nou=i_final) and (j_nou=j_final) then
                   begin
                    {am ajuns la cascaval !!}
                    Sound(400); Delay(100); NoSound;
                    PrintAt(j_final,i_final,'F'); Delay(500)
                   end
                else Drum(i_nou,j_nou,pas+1);
                Traseu[i_nou,j_nou]:=0; PrintAt(j_nou, i_nou,' ')
              end
      end
end;
procedure InitializeazaLabirint;
var b,x,y,i,j: Integer;
begin
  { se initializeaza matricea Traseu }
  for i:=1 to m do for j:=1 to n do Traseu[i,j]:=0;
  { se delimiteaza portiunea labirintului in coltul stanga-sus }
  { propunem cititorului sa modifice programul pentru a desena labirintul in
orice pozitie, de pilda in centrul ecranului ! }
  for i:=1 to m+1 do PrintAt(n+1,i,#219);
  for i:=1 to n+1 do PrintAt(i,m+1,#219);
  { la inceput labirintul are doar casute de trecere }
  for i:=1 to m do for j:=1 to n do Labirint[i,j]:=1;
  { se monteaza si afiseaza zidurile, cu mouse-ul }
  MouseInit; MouseShow;
  repeat
    MouseTData(b,x,y);
    if (b=1) and Apartine(x,y,1,1,n,m) then
       begin
         MouseHide; PrintAt(x,y,#178); MouseShow; Labirint[y,x]:=0
       end
  until (b=2) and Apartine(x,y,1,1,n,m);
  { se stabileste pozitia de start "S" ... }
  Labirint[x,y]:=1; i_initial:=y; j_initial:=x;
  MouseHide; PrintAt(x,y,'S'); MouseShow; Delay(300);
  { ... si pozitia de sosire "F" }
  repeat MouseTData(b,x,y) until (b=2) and Apartine(x,y,1,1,n,m);
  i_final:=y; j_final:=x; MouseHide; PrintAt(x,y,'F')
end;
```

```
begin
  ClrScr; InitializeazaLlabirint; Traseu[i_initial, j_initial]:=1;
  Drum(i_initial, j_initial, 2); ReadLn
end.
```

Iată, de exemplu, un moment din acest "joc", când şoricelul tocmai ajunge la caşcaval!



6. Metoda "*divide et impera*"

Prezentare generală

"***Divide et impera***" este o tehnică (recursivă) ce constă în următoarele:

- dacă problema este rezolvabilă direct, atunci ea se rezolvă
- altfel se descompune în două sau mai multe probleme mai simple, de aceeaşi natură cu problema inițială (numite subprobleme), care se rezolvă prin aceeaşi metodă;
- soluția problemei inițiale se obține prin combinarea soluțiilor subproblemelor.

De exemplu, fie $A=(a_1, a_2, ..., a_n)$ şi trebuie efectuată o prelucrare oarecare asupra elementelor sale. Mai mult, presupunem că pentru orice p, q naturale, cu $1 \le p < q \le n$, există $m \in \{p, ..., q-1\}$ astfel încât prelucrarea secvenței $\{a_p, ..., a_k\}$ se poate face prelucrând secvențele $\{a_p, ..., a_m\}$ şi $\{a_{m+1}, ..., a_q\}$. Se apelează procedura descrisă mai jos, astfel: `DivideEtImpera(1,n,α)`, în α obținându-se rezultatul final. Am notat prin ε lungimea maximă a unei secvențe $\{a_p, ..., a_q\}$ pentru care prelucrarea se poate face direct (prin procedura `Prelucreaza`). Procedura `Combina` realizează combinarea rezultatelor a două secvențe vecine. `m` este obținut prin apelul procedurii `Divide`.

Iată deci varianta recursivă a metodei "*divide et impera*":

```
procedure DivideEtImpera(p, q: Integer;  var α: ...);
begin
  if q - p < ε then Prelucreaza(p, q, α)
  else
    begin
      Divide(p, q, m);
      DivideEtImpera(p, m, β); DivideEtImpera(m+1, q, γ);
      Combina(β, γ, α)
    end
end;
```

Printre exemplele clasice care folosesc această metodă se numără: *sortarea prin interclasare; sortarea rapidă* ("*quick-sort*")*; turnurile din Hanoi* (vezi lecția 5!).

Sortarea prin interclasare

Algoritmul de **sortare prin interclasare** constituie un exemplu reprezentativ pentru folosirea metodei "*divide et impera*" în programare. Astfel, dacă avem de sortat un vector, atunci îl împărțim în două, sortăm - la fel - cele două părți ale vectorului, apoi le interclasăm. Dacă şi vectorii rezultați după împărțire sunt destul de mari (mai mult decât un singur element), atunci procedăm la împărțirea şi a acestor vectori şi tot aşa.

Astfel, vom scrie o procedură `SortInterclas(inceput,sfarsit:Integer);` care va sorta vectorul `A` între poziția `început` şi poziția `sfarsit`. Procedura va determina poziția din `mijloc` şi se va autoapela pentru `început` şi `mijloc`, apoi pentru `mijloc+1` şi `sfarsit`, după care vor interclasa cele două părți ale vectorului (vezi lecția 4!).

```
program SortarePrinInterclasare;
const max=10;
var A: array[1..max] of Integer; i,n: 1..max;
procedure Interclaseaza(start,mijloc,finis: Integer);
var B: array[1..max] of Integer; i,j,k: Integer;
begin
  k:=start; i:=start; j:=mijloc+1;
  while (i<=mijloc) and (j<=finis) do
        if A[i]<A[j] then begin B[k]:=A[i]; i:=i+1; k:=k+1 end
        else begin B[k]:=A[j]; j:=j+1; k:=k+1 end;
  if i<=mijloc then for j:=i to mijloc do begin B[k]:=A[j]; k:=k+1 end
  else for i:=j to finis do begin B[k]:=A[i]; k:=k+1 end;
  for i:=start to finis do A[i]:=B[i]
end;
procedure SortInterclas(inceput,sfarsit: Integer);
var centru: Integer;
begin
  if inceput < sfarsit then
   begin
     centru:=(inceput+sfarsit) div 2; SortInterclas(inceput,centru);
     SortInterclas(centru+1,sfarsit); Interclaseaza(inceput,centru, sfarsit)
   end
end;
begin
  Write('n='); ReadLn(n);
  for i:=1 to n do begin Write('A[',i,']='); ReadLn(A[i]) end;
  SortInterclas(1,n); for i:=1 to n do Write(A[i],','); ReadLn
end.
```

Sortarea rapidă "*quick-sort*"

Un alt exemplu îl constituie acest algoritm avansat de sortare, datorat profesorului *C. Hoare*, care foloseşte o procedură `Pozitioneaza`. Această procedură se ocupă de o anumită parte din vectorul de sortat `A`, cuprinsă între indicii `start` şi `finis`. Ea poziționează componenta de pe poziția `start` pe o anumită poziție `k`, între `start` şi `finis`, poziție pe care respectivul element va rămâne până la final, astfel încât toate elementele de pe poziții între `start` şi `k-1` să fie mai mici sau egale cu `A[k]`, iar toate elementele de pe poziții între `k+1` şi `finis` să fie mai mari sau egale cu `A[k]`.

În procedura de sortare, `Quick`, după ce s-a realizat poziționarea, în conformitate cu tehnica "*divide et impera*", se va autoapela această procedură pentru cele două părți rămase nesortate, dinainte şi de după elementul poziționat `A[k]`.

În procedura `Pozitioneaza` se compară, în mod repetat, două elemente, cele de pe pozițiile `i` şi respectiv `j` din vector. Inițial `i` este `start`, iar `j` este `finis`. La fiecare pas, dacă `A[i]>A[j]` se interschimbă `A[i]` cu `A[j]` şi se trece fie la mărirea lui `i` cu o unitate, fie la micşorarea lui `j` cu o unitate, alternativ. Astfel, până când `i` devine egal cu `j`, pozițiile `i` şi `j` se apropie, alternativ, una de alta, odată cu eventualele interschimbări necesitate de relația existentă între elementele de pe cele două poziții. Modificările lui `i` şi `j` se realizează cu ajutorul variabilei `d`, care ia pe rând valorile `0` şi `1`, iar asupra lui `i` se execută o incrementare cu `d`, iar asupra lui `j` o decrementare cu `d`.

```
program QuickSort;
const max=10;
type vector=array[1..max] of Integer; var A: vector;  i,n: Integer;
procedure Pozitioneaza(start,finis:Integer; var k:Integer; var A:vector);
  procedure Schimba(var x,y: Integer);
  var aux: Integer; begin aux:=x; x:=y; y:=aux end;
var i,j,d: Integer;
```

```
begin
     d:=0; i:=start; j:= finis;
     while i<j do
           begin
             if A[i]>A[j] then begin Schimba(A[i],A[j]); d:=1-d end;
             Inc(i,d); Dec(j,1-d)
           end;
     k:=i
end;
procedure Quick(inceput, sfarsit: Integer; var A: vector);
var k: Integer;
begin
     if inceput<sfarsit then
        begin
          Pozitioneaza(inceput, sfarsit,k,A);
          Quick(inceput,k-1,A); Quick(k+1,sfarsit,A)
        end
end;
begin
     WriteLn('Quick - sort'); Write('Dati n = '); ReadLn(n);
     for i:=1 to n do begin Write('. A[',i,'] = '); ReadLn(A[i]) end;
     Quick(1,n,A);
     WriteLn('Vectorul sortat este: ');
     for i:=1 to n do Write(A[i],', ');
     ReadLn
end.
```

7. Metoda programării dinamice

Prezentare generală

Această metodă se utilizează în problemele în care rezultatul se obține în urma aplicării - aupra unui *sistem* - a unei *politici*, adică a unei secvențe de *decizii* D_1, D_2, ..., D_n. Aplicarea deciziei D_i determină ca sistemul să evolueze din starea S_{i-1} în starea S_i, pentru orice i între 1 şi n.

În afara acestor lucruri, trebuie să fie satisfăcut **principiul optimalității (a programării dinamice)**: Dacă D_1, D_2, ..., D_n este un şir optim de decizii care duce sistemul din starea inițială S_0 în starea finală S_n, atunci trebuie îndeplinită una din următoarele condiții:

1) D_k, ..., D_n este un şir optim de decizii care duce sistemul din starea S_{k-1} în starea S_n; (*metoda înainte*)

2) D_1, ..., D_k este un şir optim de decizii care duce sistemul din starea S_1 în starea S_k; (*metoda înapoi*)

3) D_1, ..., D_k şi D_{k+1}, ..., D_n sunt şiruri optime de decizii care transformă starea sistemului din S_1 în S_k, respectiv din S_{k+1} în S_n, $\forall k=1,...,n$. (*metoda mixtă*).

Pentru a rezolva o problemă prin procedeul programării dinamice se urmăresc etapele:

- verificarea principiului de optimalitate în una din cele trei forme ale sale;
- scrierea relațiilor ce apar, corespunzătoare formei în care este verificat principiul de optimalitate şi, fireşte, redactarea programului.

De notat faptul că rezolvarea prin această tehnică a problemelor de multe ori nu este unică. Metoda este destul de bună, deoarece, ea nu generează toate posibilitățile şi o extrage pe cea optimă dintre ele.

Exemple de probleme celebre ce se soluționează cu această metodă sunt: *problema subşirului crescător de lungime maximă; determinarea costurilor drumurilor de cost minim într-un graf; înmulțirea optimă a unui şir de matrice, problema raftului.*

Problema subşirului crescător de lungime maximă

Se dă un şir (un vector) de n numere întregi. Se cere să se determine cel mai lung subşir crescător al acestuia. De exemplu, pentru vectorul $V=(4,1,8,5,8)$ de lungime $n=5$, răspunsul

este: 4,8,8. Rezolvarea problemei se face pornind de la ideea determinării, pentru fiecare element din vector, a lungimii celui mai lung subşir crescător ce se poate forma începând cu acel element. În final se va alege elementul cu care se poate forma cel mai lung subşir crescător, care se va afişa.
Fie $V[i_1]$, $V[i_2]$, ..., V[k], ..., $V[i_p]$ cel mai lung subşir care se poate forma cu elementul V[k]. Atunci V[k], ..., $V[i_p]$ este cel mai lung subşir ce se poate forma de la V[k] încolo. Aşadar, se verifică principiul programării dinamice pe varianta *metodei înainte*.

Să memorăm, folosind un vector L cu n elemente, în L[k] lungimea celui mai lung subşir crescător ce se poate forma începând cu V[k]. Vom avea relația:
$L[k]=1+\max\{L[i]; V[k] \le V[i], i \in \{k+1,...,n\}\}$, pentru orice k între n-1 şi 1, şi vom avea grijă, la fiecare pas să se păstreze această relație (vom actualiza pe L[k]).
În exemplul dat avem: L=(3,3,2,2,1). Pentru a determina acest vector s-a procedat astfel:

- pentru elementul 8 de pe ultima poziție, avem evident L[5]=1, deoarece de la el încolo se poate forma doar un singur subşir crescător de lungime 1 (elementul 8 însuşi);
- pentru elementul 5 (poztia 4) vom avea L[4]=2=1+L[5], deoarece există după el elementul 8 care este mai mare;
- pentru elementul 8 (poziția 3) vom avea cel mai lung subşir crescător începând cu el, format tot cu elementul de pe ultima poziție (8,8), deci L[3]=2=1+L[5];
- pentru elementul 1 avem L[2]=3=1+L[3];
- pentru elementul 4 avem L[1]=3=1+L[3].

Pentru a obține subşirul crescător maximal, se procedează după cum urmează:

- se caută maximul din vectorul L, precum şi poziția t pe care se află acest maxim;
- se afişează V[t];
- în mod repetat, se găseşte şi se afişează primul element $V[i] \ge V[t]$, cu L[i]=maxim-1, unde maxim este la început L[t], apoi descreşte, la fiecare afişare a lui V[i] cu câte o unitate, el indicând lungimea maximă a acelei părți din subşirul crescător maximal care mai trebuie afişată încă.

```
program SubsirCrescatorMaximal;
const max=10;
var V,L: array[1..max] of Integer; n,i,k,maxim,t: Integer;
begin
  { citirea vectorului (sirului) }
  Write('n='); ReadLn(n);
  for i:=1 to n do begin Write('V[',i,']='); ReadLn(V[i]) end;
  { se considera vectorul L }
  L[n]:=1;
  for k:=n-1 downto 1 do
      begin
        maxim:=0;
        for i:=k+1 to n do
            if (V[i]>=V[k]) and (L[i]>maxim) then maxim:=L[i];
        L[k]:=1+maxim
      end;
  maxim:=L[1]; t:=1;
  { se cauta capatul de inceput al subsirului, V[t] cu L[t] primul maxim }
  for k:=1 to n do
      if L[k] > maxim then begin maxim:=l[k]; t:=k end;
  WriteLn('Lungimea maxima = ', maxim); Write(V[t],',');
  { se afiseaza restul sirului }
  for i:=t+1 to n do
      if (V[i]>=V[t]) {!} and (L[i]=maxim-1) then
         begin Write(V[i],','); maxim:=maxim-1 end;
  ReadLn
end.
```

Problema raftului

Problema a fost propusă de profesorul *Tudor Sorin* la *Olimpiada Națională de Informatică*, Suceava 1996, sub numele de "*problema patronului*".

Se dă un vector `var x: array[1..60] of Integer` şi `S: 1..200`. Se cere să se determine un subşir $x[i_1],x[i_2],...,x[i_k]$ de elemente din `x` astfel încât suma lor să fie `S`, iar `k` să fie maxim. În problema originală, `S` era înălțimea unui raft în care patronul trebuia să aşeze orizontal nişte cărți, cu grosimile date de componentele vectorului `x`.

Exemplu: pentru `S=10` şi `x=(2,4,5,6,1,3)`, obținem soluția: `4` cărți şi anume cărțile `1,2,3` şi `4`.

Folosirea metodei "*back-tracking*" pentru a genera vectorul optim căutat se dovedeşte ineficientă în această problemă, de aceea vom folosi *metoda programării dinamice*. Vom considera un vector `y: array[0..200] of Integer` şi un vector `m: array[0..200] of set of Integer`, cu semnificațiile următoare:

- `y[i]`=numărul maxim de cărți care dau suma `i`
- `m[i]`=mulțimea indicilor cărților alese pentru a da suma `i`

Astfel, dinamic, se determină `y[i]` şi `m[i]`:

- o carte `j`, de grosime `x[j]` se ia în mulțimea `m[i]` (care formează suma `i`) doar dacă `x[j]` încape în `i` şi dacă cartea `j` nu a mai fost luată cumva în mulțimea `m[i-x[j]]`, adică într-o mulțime ce ar forma suma egală cu diferența între `i` şi grosimea `x[j]`;
- de asemenea, pentru a obține soluția optimă, cartea `j` se ia în mulțimea `m[i]`, dacă noua mulțime, care ar fi reuniunea dintre mulțimea `m[i-x[j]]` şi mulțimea cu elementul `j` ar fi mai numeroasă, adică `y[i-x[j]]+1>y[j]`.

```
program Patron;
const Nmax=60; Smax=200;
var x: array[1..Nmax] of Byte; y: array[0..Smax] of Byte;
    m: array[0..Smax] of set of Byte; n {numarul de carti},i,j,k,S: Integer;
begin
  WriteLn('Problema patronului'); WriteLn;
  { citirea datelor: inaltimea raftului, apoi grosimile cartilor }
  Write('Dati inaltimea raftului: '); ReadLn(S);
  i:=1;
  Write('Dati numarul de carti: '); ReadLn(n);
  for i:=1 to n do
      begin
        Write(' - Dati grosimea cartii nr.',i,': ');
        ReadLn(x[i])
      end;
  { initiliazarea vectorilor y si m }
  for i:=0 to S do begin y[i]:=0; m[i]:=[] end;
  for i:=1 to S do
      for j:=1 to n do
          if i-x[j] >= 0 then
             if y[i-x[j]] + 1 > y[i] then
                if not (j in m[i-x[j]]) then
                   begin
                     { numarul de elemente din multimea j este 1 +
                       al celor din multimea ce contine diferenta }
                     y[i] := 1 + y[i-x[j]];
                     { se pune cartea j in multimea ce formeaza suma i,
                       alaturi de cartile din multimea i-x[j] }
                     m[i] := m[i-x[j]]+[j]
                   end;
  WriteLn('Numarul maxim de carti: ',y[S]); WriteLn('Cartile alese sint: ');
  for j:=1 to n do
      if j in m[S] then Write(x[j],','); ReadLn
end.
```

8. Exerciții recapitulative

Utilizarea eficientă a metodelor de elaborare a algoritmilor prezentate în acest capitol, combinată cu perspicacitatea rezolvitorului şi abilitatea sa de a modela matematico-informatic realitatea înconjurătoare - iată cheia reuşitei în concursurile şcolare de informatică!

1. (Olimpiada Națională, Slatina 1995) Unor elevi li se cere să pună $n<200$ cuvinte formate doar din litere mici într-o ordine firească. De exemplu: *platon kant marx stalin havel* (ordine cronologică). Fiecare elev îi dă profesorului o succesiune de cuvinte pe care o consideră corectă. Problema pe care trebuie să o rezolve profesorul este de a puncta fiecare elev cu o notă între 1 şi n, reprezentând numărul de cuvinte plasate într-o succesiune corectă. Pentru exemplul anterior, avem următoarele secvențe date de elevi şi notele obținute:

```
marx stalin kant platon havel        3
havel marx stalin kant platon        2
havel stalin marx kant platon        1
```

Ajutați-l pe profesor !

2. Problema platourilor. Fie un tablou `a: array[0..n-1]` cu elemente de tip întreg cu proprietatea $a[0]\le a[1]\le...\le a[n-1]$. Un *platou* este o subsecvență maximală de elemente consecutive în tablou care conține aceeaşi valoare. Lungimea unui platou este numărul de elemente ce compun tabloul. Se cere să se determine lungimea `p` a celui mai lung platou.

3. Să se scrie un program care să determine pentru un vector `a` numărul platourilor (vezi problema anterioară). Definim noțiunea de *pantă* ca fiind o secvență de elemente consecutive din `a` astfel încât $a[i]=a[i-1]+1$. Să se determine panta de lungime maximă.

4. Numere pitagoreice. Să se determine, pentru un număr întreg `n` dat, toate tripletele de numere pitagoreice (a,b,c) (deci $c^2=a^2+b^2$) astfel încât $1\le a,b\le n$).

5. Problema steagului național olandez. (Bacău 1994). Se consideră `type culoare=(rosu,alb,albastru)` şi `var u: array[1..n] of culoare (n≥1)`. Să se aranjeze, prin interschimbări, vectorul `u` astfel încât întâi să avem elementele roşii, apoi cele albe, apoi cele albastre. (Elementele pot fi considerate nişte pioni ce se pot interschimba.).

6. Să se aşeze pe o tablă de şah $n\times n$ `n` "piese", câte una pe fiecare coloană, care să nu se atace între ele. O "piesă" atacă:

- precum un cal;
- precum un cal şi un rege;
- precum un cal şi o tură;
- precum un cal şi un nebun;
- precum un nebun.

7. Să se scrie un program care să caute binar un element într-un vector. Se va folosi algoritmul descris în lecția 1, care se bazează pe metoda "*divide et impera*".

8. (Alba, 1994) Se dau `n` litere mici. Să se formeze:

a) toate cuvintele posibile din `m` litere ($m\le n$), în care literele sunt ordonate alfabetic în cadrul cuvântului, iar o literă poate apare de cel mult `m` ori;

b) toate cuvintele posibile din `m` litere, în care literele sunt ordonate alfabetic în cadrul cuvântului, iar o literă se poate repeta de cel mult `k` ori.

9. (Teleorman, 1994) La un concurs sunt prezentate 5 melodii `A`, `B`, `C`, `D`, `E`. Să se afle toate posibilitățile de a le prezenta, ştiind că melodia `B` va fi interpretată înaintea melodiei `A`.

10. Să se genereze, folosind metoda "*back-tracking*" permutările de grad `n`.

11. Să se parcurgă, cu un cal, o tablă de şah cu `n` linii şi `n` coloane, în aşa fel încât prin fiecare căsuță să se treacă exact o singură dată.

12. Să se aşeze $2n-2$ nebuni pe o tablă de şah cu n^2 pătrate astfel încât nici o pereche de nebuni să nu se amenințe.

13. Să se aşeze pe o tablă de şah cu n^2 pătrate cât mai multe dame care să nu se atace între ele.

14. Scrieți o procedură recursivă şi una iterativă pentru a căuta un cuvânt într-un vector de cuvinte, care sunt puse în ordine alfabetică.
15. Se dă o bucată dreptunghiulară de tablă cu lungimea l şi lățimea h, având n găuri, de coordonate numere întregi. Să se decupeze din ea o bucată de arie maximă care nu prezintă găuri. Sunt permise doar tăieturi verticale şi orizontale. Se va utiliza metoda "*divide et impera*".
16. Pe produsul cartezian $N \times N$ se defineşte operația: $(a,b)(b,c)=(a,c)$, pentru orice a, b şi $c \in \mathbf{N}$, despre care ştim că:

a) este asociativă: $((a,b)(b,c))(c,d)=(a,b)((b,c)(c,d))$;
b) efectuarea ei necesită exact $a \times b \times c$ secunde.

Fiind date $x_1, x_2, ..., x_n \in N$, $n \geq 3$, care este timpul minim şi cel maxim în care se poate efectua produsul $(x_1,x_2)(x_2,x_3)...(x_{n-1},x_n)$?
De exemplu, pentru produsul $(7,1)(1,9)(9,3)$ avem rezultatul $(7,3)$, care se poate obține cel puțin în 48 secunde şi cel mult în 4 minute şi 12 secunde.
17. Un țăran primeşte o bucată dreptunghiulară de pământ pe care doreşte să planteze o livadă. Pentru aceasta, el va împărți bucata de pământ în $m \times n$ pătrate, având dimensiunile egale, iar în fiecare pătrat va planta un singur pom din cele patru soiuri pe care le are la dispoziție. Să se afişeze toate variantele de a alcătui livada respectând următoarele condiții:
a) Nu trebuie să existe doi pomi de acelaşi soi în două căsuțe învecinate ortogonal sau diagonal.
b) Fiecare pom va fi înconjurat de cel puțin un pom din toate celelalte trei soiuri.
Observație: țăranul are la dispoziție suficienți pomi de fiecare soi.
19. Un teren muntos are forma unei matrice cu $m \times n$ zone, fiecare zonă având o înălțime. Un alpinist pleacă dintr-o anumită zonă şi trebuie să ajungă într-o zonă maximă în altitudine. Dintr-o zonă, alpinistul se poate deplasa diagonal sau ortogonal, într-una din zonele (căsuțele) alăturate, doar urcând sau mergând la acelaşi nivel. Poate el ajunge într-unul din vârfuri? Dacă da, arătați toate soluțiile problemei.
20. Attila şi regele. Un cal şi un rege se află pe o tablă de şah. Unele câmpuri sunt "arse", pozițiile lor fiind cunoscute. Calul nu poate călca pe câmpuri "arse", iar orice mişcare a calului "arde" câmpul pe care se duce. Să se afle dacă există o succesiune de mutări permise (cu restricțiile de mai sus) prin care calul să ajungă la rege şi să revină la poziția inițială. Poziția inițială a calului, precum şi poziția regelui sunt considerate nearse.

Lecția7
ALGORITMI REFERITORI LA GRAFURI

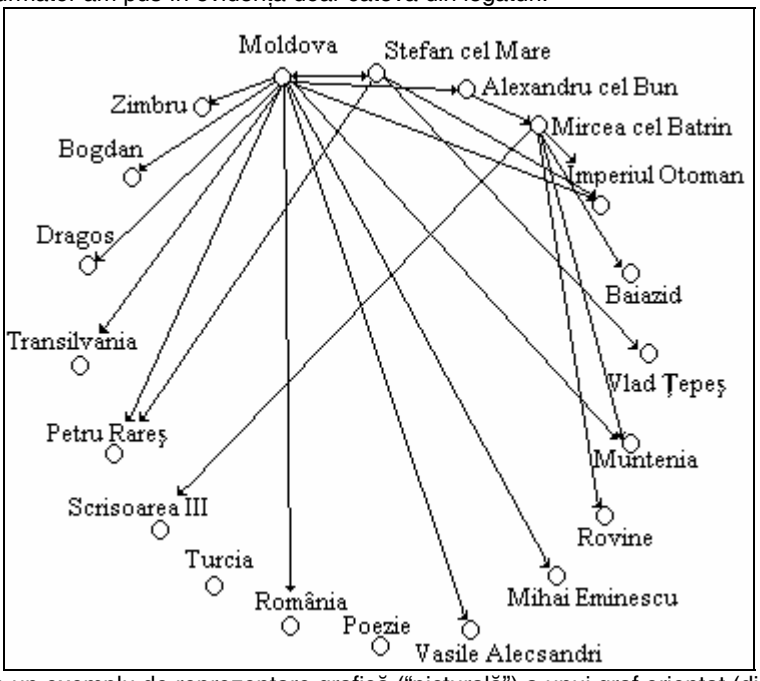
1. Noțiuni introductive

Necesitate

Să zicem că dorim să realizăm o mică enciclopedie. Dispunem de un dicționar de *articole* (cuvinte şi nume proprii), despre care avem mai multe informații (de pildă definiții pentru cuvinte, texte explicative pentru unele personalități, hărți ale unor zone geografice). Aceste articole sunt *legate* între ele, în sensul că, odată ajunşi să studiem un anumit articol, ni se pune la dispoziție o *listă de trimiteri la articole înrudite*, pe care le putem numi articole *vecine*. Nu e obligatoriu ca, dacă un articol `i` face trimitere la un articol `j`, să avem şi trimiterea inversă, de la `j` la `i`.

Cum vom memora articolele şi legăturile dintre ele? Cea mai indicată structură de date, într-o asemenea situație, este *graful*. Un graf va fi o rețea de noduri, legate prin curbe orientate sau nu (numite *arce*, respectiv *muchii*)..

În exemplul următor am pus în evidență doar câteva din legături.



Desenul este un exemplu de reprezentare grafică ("picturală") a unui graf orientat (digraf).

Definiția grafului

Prin **graf** înțelegem o pereche ordonată `G=(V,E)`, unde `V` este o mulțime nevidă, finită, ale cărei elemente se numesc **vârfuri** (**noduri**), iar `E` este o submulțime a lui $P_2(V)$, adică este formată din perechi neordonate de vârfuri: `{v,w}∈E, cu v,w∈V`. Elementele mulțimii `E` se numesc **muchii**. Dacă `e={v,w}∈E` , atunci se mai notează simplu `e=vw` şi se spune că în graful `G` vârfurile `v` şi `w` sunt *adiacente*, sau că muchia `e` este *incidentă* cu vârfurile `v` şi `w`, sau că `v` şi `w` sunt *extremitățile* muchiei `e`, sau că `v` şi `w` sunt *vecine* în graful `G`.

Deci un graf poate fi considerat ca o mulțime de vârfuri, dintre care unele sunt unite între ele două câte două. Un graf `G` poate fi desenat în plan astfel: vârfurile se reprezintă prin puncte sau

mici cerculeţe, iar muchiile prin curbe sau linii care le unesc. Vârfurile pot fi etichetate, ca în exemplul din figura anterioară.

Numim **vecinătatea** unui vârf v mulțimea vecinilor săi, iar numărul de elemente ale acestei mulțimi se numeşte **gradul** vârfului v în graful G: $N_G(v)=\{w\in V; vw\in E\}$, $d_G(v)=|N_G(v)|$.

Un vârf cu gradul egal cu 1 se numeşte *vârf terminal* al grafului, iar un vârf cu gradul egal cu zero (fără vecini) se numeşte *vârf izolat* al grafului respectiv.

Teoremă: *Dacă graful* $G=(V,E)$ *are* m *muchii şi vârfurile* $v_1, v_2, ..., v_n$, *atunci:*

$$d_G(v_1)+d_G(v_2)+...+d_G(v_n)=2\cdot m.$$

Demonstrație. Fiecare muchie vw a grafului G are două extremități v şi w, ea contribuind cu o unitate la $d_G(v)$ şi $d_G(w)$. Deci suma gradelor grafului este egală cu dublul numărului de muchii. În particular, suma gradelor este un număr par.

Subgrafuri

Un **graf parțial** al unui graf $G=(V,E)$ este un graf $H=(V,E')$, care are aceeaşi mulțime de vârfuri cu G, dar nu neapărat toate muchiile: $E'\subseteq E$. Un **subgraf** al unui graf $G=(V,E)$ este un graf $H=(V',E')$, unde $V'\subseteq V$, iar muchiile din mulțimea E' sunt toate muchiile din E, care au ambele extremități în mulțimea de vârfuri V'. Deci un subgraf al unui graf G este fie G însuşi, fie se obține din G prin suprimarea unor vârfuri şi a tuturor muchiilor incidente cu aceste vârfuri. Vom spune că subgraful H este *indus* sau *generat* de mulțimea de vârfuri V' în graful G. Un graf cu n vârfuri pentru care oricare două vârfuri sunt adiacente se numeşte **graf complet** de ordin n şi se notează prin K_n. Numărul de muchii ale lui K_n este $m=C_n^2$.

Conexitate

Un **lanț** este un şir de vârfuri $L=\{v_0, v_1, ..., v_r\}$, cu proprietatea că oricare două vârfuri vecine sunt adiacente, adică $v_iv_{i+1}\in E, \forall i=0,...,r-1$. Vârfurile x_0 şi x_r se numesc *extremitățile lanțului* L, iar numărul r se numeşte *lungimea* acestui lanț.

Dacă vârfurile $v_0, ..., v_r$ sunt distincte două cîte două, atunci L se numeşte **drum** sau **lanț elementar**. Un lanț (elementar) pentru care extremitățile sunt adiacente se numeşte **ciclu** (**elementar**).

Un graf G se numeşte **conex** dacă pentru orice pereche de vârfuri $\{x,y\}$, cu $x\neq y$ există un lanț de la x la y. Orice graf poate fi partiționat unic în unul sau mai multe subgrafuri conexe ale sale, numite **componente conexe**. În general, o componentă conexă C a unui graf G se defineşte ca fiind un subgraf conex al lui G, care este maximal în raport cu această proprietate, adică nu există nici un lanț în G care să unească un vârf din C cu un vârf care nu aparține lui C.

Tipuri de grafuri

Dacă mulțimea de vârfuri V a grafului $G=(V,E)$ poate fi partiționată în două mulțimi V_1 şi V_2, astfel încât să nu existe muchii între nodurile lui V_1 sau între nodurile lui V_2, ci doar muchii xy cu $x\in V_1$, $y\in V_2$, atunci graful G se notează $G=(V_1,V_2;E)$ şi se numeşte **graf bipartit**.

Dacă între nodurile unui graf apar mai multe muchii, atunci avem de a face cu un **multigraf**.

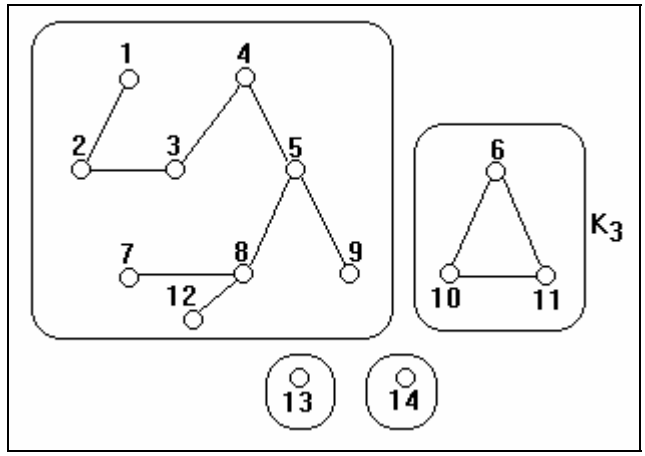
Un ciclu elementar al unui graf G care trece prin toate vârfurile grafului se numeşte **ciclu hamiltonian**. Un graf care are un ciclu hamiltonian se numeşte **graf hamiltonian**.

Prin **graf eulerian** se înțelege un graf *G* = (*V, E*) care are un **lanț eulerian** închis, cu muchii distincte, adică un lanț care trece prin toate muchiile grafului şi se întoarce în vîrful de unde a plecat.

Exemplu: În figura de mai jos este reprezentat graful $G=(V,E)$, cu $V=\{1,2,...,14\}$, iar $E=\{[1,2],[2,3],[3,4],[4,5],[5,8],[5,9],...,[13,14]\}$.

Avem $N_G(8)=\{5,7,12\}$, deci $d_G(8)=3$. $(1,2,3,4,5,8,12)$ este un drum în acest graf, iar $(8,10,11)$ este un ciclu. $1,7,12$ şi 9 sunt vârfuri terminale, iar 13 şi 14 sunt vârfuri izolate. Graful are patru componente conexe, din care una este un subgraf egal cu K_3. Graful nu este

hamiltonian şi nici eulerian, dar componenta conexă cu vârfurile 6, 10 şi 11 este un graf atât hamiltonian, cât şi eulerian.



Teoremă: *Dacă* G=(V,E) *este un graf cu* n≥3 *vârfuri, astfel încât* $d_G(v) \geq n/2$, $\forall v \in V$, *atunci* G *este hamiltonian.*

Teoremă: *Un graf fără vârfuri izolate este eulerian dacă şi numai dacă este conex şi gradele vârfurilor sunt pare*.

Digrafuri. Conexitate în digrafuri

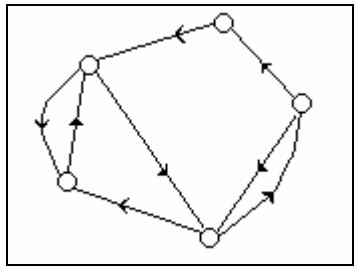
Un **digraf** (**graf orientat**) este o pereche ordonată D=(V,A), în care V este o mulțime de vârfuri (noduri), iar A este o mulțime de **arce**, adică nişte muchii *orientate*, deci perechi ordonate (u,v), cu u, v ∈ V.

În mod similar noțiunii de drum în graf, se defineşte noțiunea de drum într-un digraf, ținându-se cont de orientare. Avem, de asemenea, următoarele definiții:

- D este **tare conex** dacă ∀u,v∈V, există drum de la u la v în D.
- D este **unilateral conex** dacă ∀u,v∈V, există drum de la u la v sau de la v la u în D.
- D este **conex** dacă graful suport al său (deci graful obținut din D renunțând la orientarea arcelor) este conex.

Çi în digrafuri pot fi definite - în mod cu totul similar grafurilor - noțiuni ca *drum hamiltonian* sau *eulerian* etc..

Exemplu: Digraful din figura următoare este tare conex.



Reprezentarea grafurilor

Grafurile se pot reprezenta:

- prin **matrice de adiacență**: $A_{n \times n}$, unde n este numărul de vârfuri, cu $A_{ij}=1$, dacă $ij \in E$, respectiv 0, în caz contrar:

```
var A: array[1..max, 1..max] of 0..1;
```

Adăugarea muchiei ij înseamnă: A[i,j]:=1; A[j,i]:=1.

- prin **liste de adiacență**: pentru fiecare nod se memorează lista vecinilor săi:

Pentru acest mod de reprezentare se foloseşte un tip de date numit *referință* pe care îl vom prezenta în lecția următoare:

```
type lista = ^nod;
     nod = record
             info: 1..max;
             urm: lista
           end;
var Graf: array[1..max] of lista;
    p: lista;
```

Adăugarea muchiei ij înseamnă:

```
begin
   New(p); p^.info:=j; p^.urm:=Graf[i];
   Graf[i]:=p;
   New(p); p^.info:=i; p^.urm:=Graf[j];
   Graf[j]:=p
end;
```

Program demonstrativ

Revenind la problema cu mini-enciclopedia, dăm mai jos un program în care memorăm articolele sub forma unor înregistrări care cuprind două câmpuri, unul reprezentând termenul propriu-zis, iar cel de al doilea definiția sa (aici apare acelaşi text, exemplul este pur didactic). Relațiile dintre articole sunt memorate într-o matrice de adiacență constantă, care este asociată digrafului articolelor.

```
program Relatii;
uses Crt;
const max=20;
type  graf=array[1..max,1..max] of Integer;
      S40=String[40];
      informatie=record articol: S40; explicatie: String end;
const denumire: array[1..max] of S40 =
      ('Moldova','Stefan cel Mare', 'Alexandru cel Bun',
       'Mircea cel Batrin', 'Imperiul Otoman', 'Baiazid',
       'Vlad Tepes','Muntenia', 'Rovine', 'Mihai Eminescu',
       'Vasile Alecsandri','Poezie', 'Romania','Turcia',
       'Scrisoarea III','Petru Rares', 'Transilvania','Dragos',
       'Bogdan','Zimbru');
G: graf =
((0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1),
 (1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0),
 (1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0),
 (1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0),
 (0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0),
 (1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0),
 (1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0),
 (0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0),
 (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0),
 (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0),
 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0),
 (1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1),
 (1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0),
```

```
  (0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0),
  (1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0),
  (1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0),
  (1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1),
  (1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1),
  (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0)
);
var A: array[1..max] of informatie; Cod: array[1..max] of 1..max;
    i,j: 1..max; k:0..max;
    optiune,optiune1: 0..max; raspuns:String[2]; fis: Text;
begin
  TextColor(White); TextBackground(Black); ClrScr;
  { generare informatii }
  for i:=1 to max do
      with A[i] do
        begin
          articol:=denumire[i];
          explicatie:='text explicativ '+ 'asociat articolului '+articol
        end;
  repeat
    ClrScr; TextBackground(Blue); WriteLn('Selectati : ');
    TextBackground(Black); WriteLn('0=stop');
    for i:=1 to max do WriteLn(i,'=',A[i].articol);
    TextBackground(Green); Write('Optiunea dvs. = ');
    TextBackground(Black); ReadLn(optiune); optiune1:=optiune;
    if optiune1<>0 then
       repeat
         ClrScr; TextColor(Yellow);
         with A[optiune] do
              begin
                WriteLn(articol);
                for i:=1 to Length(articol) do Write('-');
                WriteLn
              end;
         WriteLn(A[optiune].explicatie); TextColor(LightGreen); WriteLn;
         WriteLn('Articole inrudite:'); WriteLn; TextColor(White);
         WriteLn('0=meniu principal'); k:=0;
         for i:=1 to max do
             if G[optiune,i]=1 then
                begin
                  k:=k+1; Cod[k]:=i; WriteLn(k,'.',A[i].articol)
                end;
         TextColor(LightRed);
         repeat
           Write('Articolul nou = '); ReadLn(optiune)
         until optiune in [0..k];
         TextColor(White); optiune:=Cod[optiune]
       until optiune=0
  until optiune1=0;
  TextColor(7); ClrScr
end.
```

Mai jos este figurat un moment din execuția acestui program, după ce, din meniul principal, a fost selectat cuvântul "*Çtefan cel Mare*". Este afişat termenul împreună cu explicația asociată lui, apoi o listă de cuvinte înrudite (tocmai vecinii vârfului "*Çtefan cel Mare*" din digraf). Tastându-se '0' se va reveni în meniul principal, iar dacă se tastează un număr de ordine (între 1 şi 4), acesta corespunde unui articol din vectorul de articole A, corespondența fiind realizată prin intermediul vectorului Cod. Astfel, dacă se tastează 3 se trece la articolul "*Vlad æepeş*" (cu Cod[3]=7=numărul de ordine al lui "Vlad æepeş" în vectorul A, în acel moment).

```
Stefan cel Mare
------------------
text explicativ asociat articolului Stefan cel Mare
Articole înrudite:
0 = meniu principal
```

```
1. Moldova
2. Imperiul Otoman
3. Vlad Tepes
4. Petru Rares
Articol nou = _
```

2. Explorarea grafurilor

Dându-se un graf (sau un digraf) prin matricea sa de adiacență, să se exploreze graful (digraful) pornind dintr-un anumit nod. Trecerea de la un nod la altul este perrmisă doar dacă între cele două noduri există muchie (respectiv arc). Este clar că nu vor putea fi vizitate decât acele noduri la care se poate ajunge, pe un drum, de la nodul de start.
Folosind procedurile de explorare, se cere să se determine şi componentele conexe ale grafului.

Explorarea în lărgime

La explorarea în lărgime, după vizitarea nodului inițial, se explorează toate nodurile adiacente lui, se trece apoi la primul nod adiacent şi se explorează toate nodurile adiacente acestuia, etc..

Mai precis, explorarea în lărgime continuă întotdeauna cu primul nod vizitat care mai are muchii (arce) neexplorate. Pentru a realiza acest lucru se foloseşte o coadă `Q`, în care se înscriu nodurile în ordinea în care sunt întâlnite.

Coada este o structură de date asemănătoare stivei. Se memorează sub forma unui vector la care operațiile de introducere se fac prin spate (la coada cozii), iar extragerea (în vederea prelucrării) în față (în capul cozii), spre deosebire de stivă. Funcționează deci, ca orice coadă la pâine! Aşadar, pentru a o implementa, în afară de vectorul ce cuprinde elementele cozii, vom folosi şi două variabile întregi, care să indice poziția primului element *considerat* în coadă şi, respectiv, a ultimului. Am spus "*considerat*", deoarece când vom adăuga un element în coadă vom acționa asupra indicelui de sfârşit, iar când vom elimina un element, asupra celui de început.

Mecanismul stivei se numeşte "*ultimul sosit - primul servit*", al cozii este "*primul sosit - primul servit*". Asupra stivelor şi cozilor - care se pot implementa şi altfel decât cu vectori - vom reveni în capitolul următor, odată cu luarea în discuție a tipului referință.

Algoritmul este următorul:

```
Explorare în lărgime(s) înseamnă:
început
   marchează s;
   afisează s;
   pune s în coada Q;
   cât timp coada Q nu este vidă execută
       început
         fie v = primul nod din coada Q;
         pentru fiecare nod w adiacent lui v execută
         dacă w nu este marcat atunci
              început
                 marchează w;
                 afisează w;
                 pune w în coada Q
              sfârsit
       sfârsit
sfârsit.
```

Explorarea în adâncime

La explorarea în adâncime se marchează vizitarea nodului inițial, după care se parcurge în adâncime, recursiv, fiecare nod adiacent cu el. (Iterativ, algoritmul s-ar implementa folosind o stivâ.).

Algoritmul este următorul:

```
Eplorare în adâncime(s) înseamnă:
început
   dacă s nu este marcat atunci
        început
           marchează s;
           afisează s;
           pentru fiecare nod v adiacent lui s execută
           Explorează în adâncime(v)
        sfârsit
sfârsit.
```

Determinarea componentelor conexe

Pentru a determina componentele conexe ale unui graf se poate folosi oricare din cele două metode de explorare a grafurilor dintr-un vârf. Astfel, se alege un vârf nemarcat încă şi se explorează graful, începând cu acest vârf. După terminarea explorării, prima componentă conexă este deja afişată şi se va trece la primul vârf nemarcat întâlnit şi se va continua de acolo explorarea. Procedeul se repetă pînă nu mai există vârfuri nemarcate

```
Determină componente conexe înseamnă:
început
   NrComponente:=0;
   repetă
     determină primul nod v nemarcat;
     dacă v există atunci
          început
             Explorează(v);
             treci la o linie nouă;
             NrComponente:=NrComponente+1
          sfârsit
     până când v nu mai există;
     afisează numărul de componente conexe NrComponente
sfârsit.
```

Program demonstrativ

```
program ExplorareGrafuri;
const max=10;
var G: array[1..max,1..max] of 0..1; { matricea de adiacenta a grafului }
    n: Integer; { numarul de virfuri ale grafului }
    Q: array[1..max] of Integer; { coada folosita de explorarea in largime }
    cap, fin: Integer; { capul si coada cozii } Marcat: array[1..max] of
Boolean; { vectorul marcarilor vârfurilor vizitate }
    i,j, NrComponente: Integer; {numarul de componente conexe ale grafului}
procedure Init;
begin { demarcheaza toate vârfurile grafului }
  for i:=1 to n do Marcat[i]:=False
end;
```

```
procedure ExploreazaInLargime(s: Integer);
var v, w: Integer;
begin
   cap:=1; fin:=1; Q[fin]:=s;
   Marcat[s]:=True; Write(s,',');
   while cap <= fin do
     begin { se scoate primul nod din coada: v }
        v:=Q[cap]; cap:=cap+1;
        for w:=1 to n do
            if (not Marcat[w]) and (G[v,w]=1) then
               begin
                 {toti vecinii sai w nemarcati se marcheaza,
                  afiseaza si se pun in coada }
                 Marcat[w]:=True; Write(w,',');
```

```
                fin:=fin+1; Q[fin]:=w
              end
     end
end;
```

```
{$S-}
procedure ExploreazaInAdincime(s: Integer);
var v: Integer;
begin
{daca s nu e marcat, se marcheaza, afiseaza si se exploreaza
recursiv in adâncime, începând cu toti vecinii v ai sai }
  if not Marcat[s] then
     begin
       Marcat[s]:=True; Write(s,',');
       for v:=1 to n do
           if G[s,v]=1 then ExploreazaInAdincime(v)
     end
end;
{$S+}
```

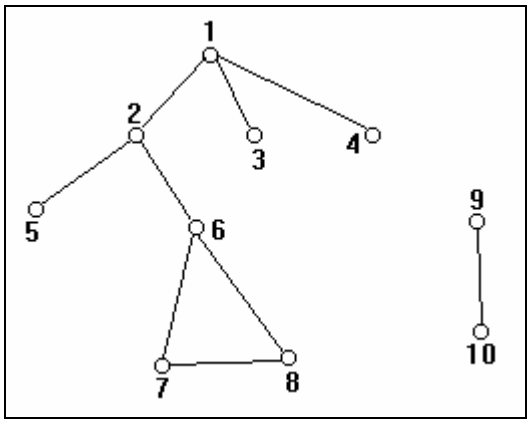
```
procedure DeterminaComponenteConexe(tip: Integer);
var v: Integer;
begin
  NrComponente:=0; WriteLn;
  repeat
    { determina primul nod v nemarcat }
    v:=1;
    while Marcat[v] and (v<=n) do v:=v+1;
    if v<>n+1 then
       begin
         case tip of
              1: ExploreazaInLargime(v);
              2: ExploreazaInAdincime(v)
         end;
         WriteLn; NrComponente:=NrComponente+1
       end
  until v=n+1; WriteLn;
  WriteLn('Sunt ',NrComponente,' componente conexe.');
  ReadLn
end;
```

```
begin
 Write('Nr. virfuri = '); ReadLn(n); { citirea grafului }
 for i:=1 to n-1 do
     begin
       G[i,i]:=0;
       for j:=i+1 to n do
           begin
             Write('G[',i,',',j,']='); ReadLn(G[i,j]);
             G[j,i]:=G[i,j] { acest lucru e valabil doar la grafuri;
                              la digrafuri trebui luat si j:=1 to n }
           end
     end;
     Init; WriteLn('Explorare in largime din nodul 1: ');
     ExploreazaInLargime(1); ReadLn;
     Init; WriteLn('Explorare in adâncime din nodul 1: ');
     ExploreazaInAdincime(1); ReadLn;
     Init; WriteLn; WriteLn('Determinarea componentelor conexe (largime):');
     DeterminaComponenteConexe(1);
     Init; WriteLn('Determinarea componentelor conexe (adâncime):');
     DeterminaComponenteConexe(2)
end.
```

Pentru graful din figura de mai jos, rezultatul programului este următorul:

```
Explorare în lărgime din nodul 1: 1,2,3,4,5,6,7,8.
Explorare în adâncime din nodul 1: 1,2,5,6,7,8,3,4.
```

```
Componente conexe (lărgime): 1,2,3,4,5,6,7,8 si 9,10.
Sunt 2 componente conexe.
Componente conexe (adâncime): 1,2,5,6,7,8,3,4 si 9,10.
Sunt 2 componente conexe.
```

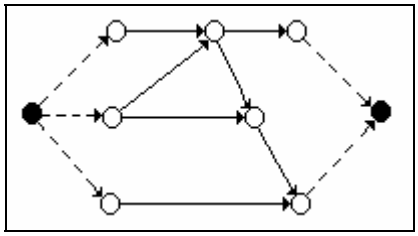


3. Drumuri de cost minim în grafuri

Probleme economice ce conduc la probleme de drumuri în (di)grafuri
Enunțăm mai jos două probleme economice diferite, dar a căror soluții se bazează pe rezolvarea unor probleme celebre de drumuri în (di)grafuri.

1. În cadrul unei întreprinderi industriale se pune problema stabilirii variantei tehnologice optime de prelucrare a unui produs. Fiecare variantă tehnologică este constituită dintr-o succesiune prestabilită de activități. Dacă se cunoaşte costul parțial de producție al fiecărei activități în parte, se cere să se determine o variantă tehnologică căreia îi corespunde un cost total de producție minim.
2. Un număr oarecare de produse trebuie prelucrate corespunzător unor relații de ordine stabilite conform necesităților producției: prelucrarea fiecărui produs poate precede prelucrarea altor produse. Se cere să se cerceteze dacă e posibilă prelucrarea produselor ținând cont de aceste relații de ordine; dacă da, se cere să se determine succesiunea în care se poate face prelucrarea.

Pentru a ne convinge că aceste probleme se reduc la probleme de teoria grafurilor, să încercăm modelarea lor matematică.
1. În cazul primei probleme, să notăm prin `n` numărul tuturor activităților, iar prin C_i costul de producție al activității `i`, `i` de la `1` la `n`. Vom reprezenta activitățile prin nişte vârfuri ale unui graf `G=(V,E)`, iar fiecare trecere de la o activitate `i` la o activitate `j` (trecere ce ar putea face parte dintr-o variantă tehnologică de producție) să o reprezentăm printr-un arc de la nodul `i` la nodul `j` în graful `G`, care va fi, deci, un graf orientat (digraf). Presupunem că orice activitate se execută o singură dată, fapt ce elimină posibilitatea ca `G` să aibă cicluri.
Putem extinde acest graf orientat, introducând un vârf etichetat cu "`0`" "înaintea" tuturor celorlalte, adică punem arc de la acest nou vârf la orice vârf din graf în care nu intră arce; adăugăm, apoi, încă un vârf, etichetat cu `n+1`, "*după*" toate celelalte vârfuri din care nu iese nici un arc. Vom pune arce între toate aceste vârfuri şi vârful `n+1`. Vom atribui apoi fiecărui arc `(ij)` costul de valoare C_j, considerând $C_{n+1}=0$.
Iată un exemplu:



2. Pentru problema a doua, să dăm un exemplu: fie de prelucrat produsele P_1, P_2, P_3, P_4, P_5, în care: P_2 precede pe P_1, P_4 şi P_5; P_3 precede pe P_2 şi P_4; P_4 precede pe P_1 şi P_5, iar P_5 precede pe P_1. Se poate observa uşor că putem prelucra cele cinci produse în ordinea P_3, P_2, P_4, P_5, P_1, relațiile de ordine anterioare respectându-se.
De fapt, dacă vom nota activitățile de prelucrare a unui produs P_i printr-un vârf i al unui graf, atunci faptul că prelucrarea lui P_i precede prelucrarea lui P_j va fi notată printr-un arc de la nodul i la nodul j. Obținem, aşadar, un digraf, iar problema economică se reduce la a găsi un drum hamiltonian în acest digraf.
Astfel, pentru a determina varianta tehnologică optimă (în sensul minimizării cheltuielilor de producție), va trebui să determinăm un drum de cost minim de la nodul 0 la nodul $n+1$.

Algoritmul lui Dijkstra
Prima problemă, cu caracter economic, prezentată se reduce la următoarea problemă de teoria grafurilor: *dându-se un (di)graf* $G=(V,E)$ *şi o funcție de cost* $C:E\to R$ *ataşată muchiilor, se cere să se determine drumurile minime (de cost minim) de la un nod* i_0 *la toate nodurile din graf, precum şi costurile acestor drumuri.*

Pentru rezolvarea problemei se foloseşte metoda "*greedy*": se selectează nodurile grafului, unul câte unul, în $n-1$ paşi, în ordinea crescătoare a costului drumului de la nodul de start la ele, într-o mulțime care inițial conține doar nodul de start.
În implementarea algoritmului se foloseşte un vector `Anterior` (cu legături de tip "tata") cu semnificația: dacă `Anterior[i]=k` atunci `k` este nodul anterior nodului `i` pe drumul minim de la i_0 (nodul de start) la nodul `i`.

De asemenea, se utilizează doi vectori cu `n` componente, `d` şi `Selectat`:

- `d[i]`=costul minim al drumului de la i_0 la `i`;
- `Selectat[i]=True` ⇔ nodul `i` este selectat.

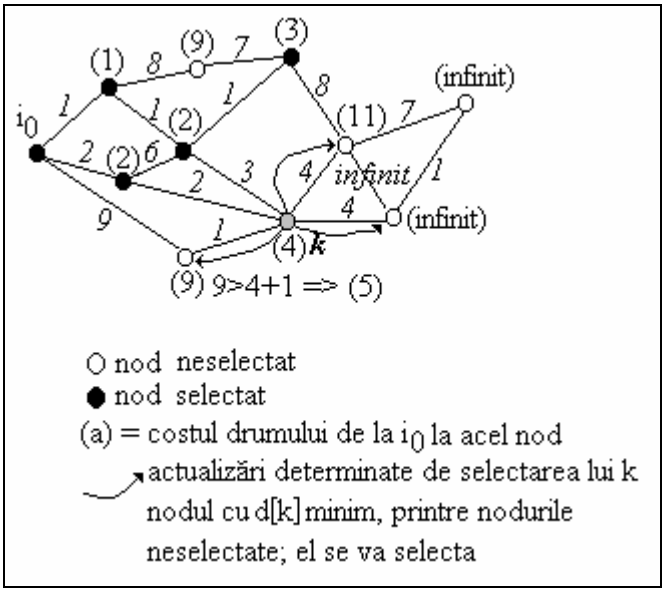
Algoritmul este:

- La început, se selectează i_0.
- La fiecare pas `p` din cei `n-1` paşi:
  - se caută nodul `i` neselectat cu `d[i]` minim, fie acesta `k` şi se selectează;
  - se actualizează vectorul `d` pentru acele noduri `i` neselectate, dar pentru care fostul `d[i]` este mai mare decât `d[k]`+costul muchiei `(i,k)`; astfel `d[i]` devine `d[k]+C[k,i]`, iar `Anterior[i]:=k`. Aşadar, drumul de la i_0 la `i` are ca nod intermediar (exact înainte de `i`) pe nodul `k`.
- La sfârşit, folosind vectorul `Anterior`, se afişează drumurile de la i_0 la fiecare nod `i` al grafului, precum şi costurile acestor drumuri.

```
program Dijkstra;
const max=10; infinit = 1000;
var C: array[1..max,1..max] of Integer; { matricea costurilor }
    i0: Integer; { nodul de start }
    d: array[1..max] of Integer;
    { d[i] = costul drumului de la i0 la i, la un moment dat }
    Selectat: array[1..max] of Boolean;
    { Selectat[i]=True <=> nodul i este in multimea nodurilor parcurse }
    Anterior: array[1..max] of Integer;
```

```
    { Anterior[i] = nodul anterior nodului i, pe drumul de la i0 la i }
    min: Integer; pas,i,j,k,n: 1..max; gata: Boolean;
{$S-}
procedure Drum(i: Integer);
{ afiseaza drumul de la i0 la i, recursiv }
begin
     if Anterior[i]<>0 then
        begin Drum(Anterior[i]); Write(',',i) end
     else Write(i)
end;
{$S+}
begin { citirea grafului }
  Write('Nr. virfuri = '); ReadLn(n);
  for i:=1 to n-1 do
      begin
        C[i,i]:=0;
        for j:=i+1 to n do {pentru digrafuri se va pune "for j:=1 to n do"}
            begin
              Write('C[',i,',',j,']='); ReadLn(C[i,j]);
              if C[i,j]=0 then C[i,j]:=infinit;
              C[j,i]:=C[i,j] { nu e valabila la digrafuri ! }
            end
      end;
  Write('i0 ='); ReadLn(i0); { citirea nodului de start }
  for i:=1 to n do
      begin
       { la inceput nici un nod nu e selectat }
        Selectat[i]:=False;
        { se considera costul drumului de la i0 la i
          ca fiind costul muchiei de la i0 la i }
        d[i]:=C[i0,i];
        { daca acesta e un numar finit, atunci nodul
          anterior lui i se considera i0, altfel 0 }
        if d[i]<infinit then Anterior[i]:=i0
        else Anterior[i]:=0
      end;
      { se selecteaza nodul de start i0, care nu are
        anterior (0), iar costul drumului de la i0 la el
        insusi este 0 }
  Selectat[i0]:=True; Anterior[i0]:=0; d[i0]:=0;
  for pas:=1 to n-1 do
      begin
        { printre nodurile neselectate, se cauta cel aflat la
          distanta minima fata de i0 si se selecteaza }
        min:=infinit;
        for i:=1 to n do
            if (not Selectat[i]) and (d[i]<min) then
               begin min:=d[i]; k:=i end;
        {k este nodul selectat, deci se adauga multimii}
        Selectat[k]:=True;

        for i:=1 to n do
            if (not Selectat[i]) and (d[k]+C[k,i]<d[i]) then
               begin d[i]:=d[k]+C[k,i]; Anterior[i]:=k end
      end;
  { se afiseaza drumurile de la i0 la fiecare nod i si costurile lor }
  for i:=1 to n do
      begin
        WriteLn('Drumul minim de la ',i0,' la ',i,':'); Drum(i);
        WriteLn; WriteLn('Costul sau este: ',d[i]); WriteLn
      end;
  ReadLn
end.
```



Determinarea costurilor tuturor drumurilor minime
Într-un graf dat prin matricea `C` *a costurilor ataşate muchiilor, se cere să se determine lungimea drumurilor de cost minim între oricare două vârfuri.*

Pentru rezolvare se foloseşte *metoda programării dinamice*: fie `i` şi `j` două vârfuri ale grafului; dacă drumul minim de la `i` la `j` trece prin vârful intermediar `k`, atunci drumurile de la `i` la `k` şi de la `k` la `j` sunt minime.

Vom construi pe rând, pornind de la matricea C^0=C, recurent, un şir de `n` matrici C^k, matricea C^k obținîndu-se folosind matricea C^{k-1}. Prin C^k`(i,j)` vom înțelege lungimea minimă între `i` şi `j`, drumul trecând numai prin prin primele `k` vârfuri.

Elementele matricei C^k se determină astfel: pentru fiecare pereche de vârfuri `(i,j)` lungimea drumului optim se calculează ca minim între drumul optim deja găsit (drum care poate trece doar prin primele `k-1` vârfuri) şi suma între lungimile drumurilor optime care legau vârful `i` de vârful `k` şi respectiv vârful `k` de vârful `j`, aceste drumuri trecând numai prin primele `k-1` vârfuri.

Aşadar, putem scrie relațiile:

$$C^k(i,j) = \begin{cases} C(i,j), & k = 0 \\ min(C^{k-1}(i,j), C^{k-1}(i,k) + C^{k-1}(k,j)), & k \neq 0 \end{cases}$$

Deoarece C^k`(i,k)`=C^{k-1}`(i,k)` şi C^k`(k,j)`=C^{k-1}`(k,j)`, înseamnă că toate calculele se pot face utilizând matricea `C`.

```
program DrumuriMinime;
{ programare dinamica }
const infinit=10000; max=10;
{ C[i,j] = infinit sau -1, daca (i,j) nu este muchie = 0, pt. i=j
         = c, pt. i<>j, iar (i,j) este muchie in graf }
var C: array[1..max,1..max] of Integer; n,i,j,k: 1..max;
begin
```

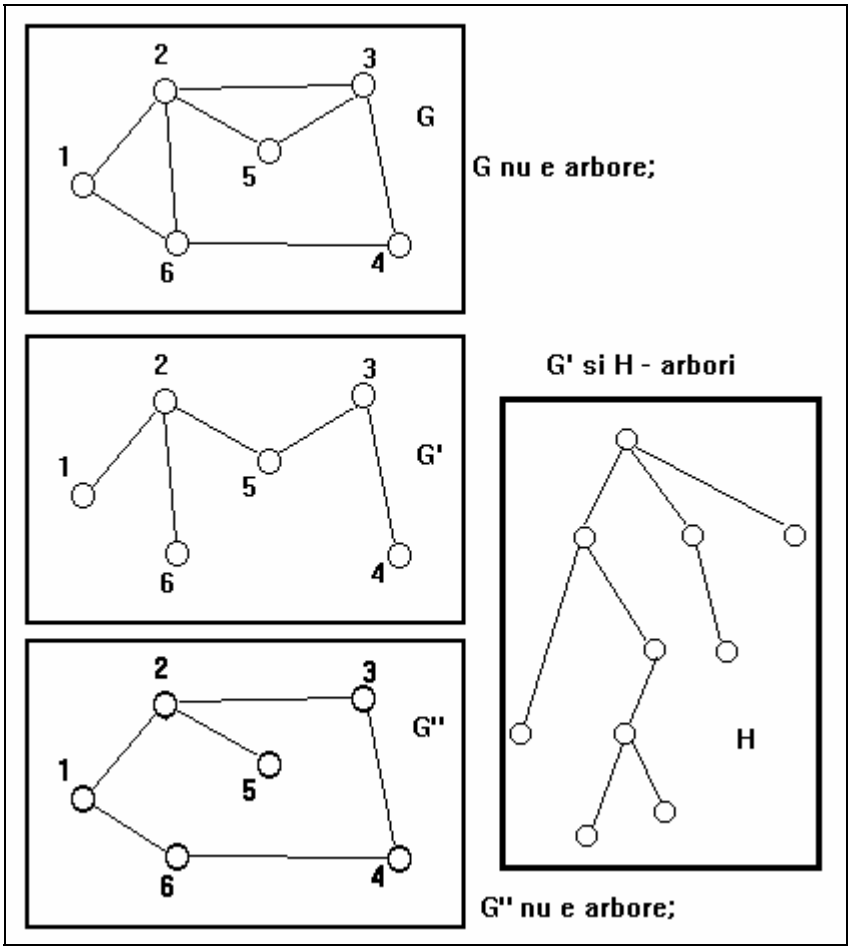
```
  Write('n='); ReadLn(n); { citirea grafului }
  for i:=1 to n-1 do
      for j:=i+1 to n do
          begin
            Write('C[',i,',',j,']='); ReadLn(C[i,j]);
            if C[i,j]=-1 then C[i,j]:=infinit;
            C[j,i]:=C[i,j]
          end;
  for i:=1 to n do C[i,i]:=0;
      { intre i si i nu este muchie }
  for k:=1 to n do
      for i:=1 to n do
          for j:=1 to n do
              if C[i,j] > C[i,k] + C[k,j] then
                 C[i,j] := C[i,k] + C[k,j];
  for i:=1 to n do
      for j:=i to n do
          WriteLn('Cost drum minim de la ',i,' la ',j,' = ',C[i,j]);
  ReadLn
end.
```

4. Arborele parțial de cost minim

Definiții

Se numeşte **arbore** orice graf conex şi fără cicluri. **Arbore parțial** al unui graf conex `G` este un graf parțial al lui `G`, care este arbore. De pildă pentru graful `G` din figura următoare, graful său parțial `G'` este arbore parțial, dar `G''` nu este, deoarece conține un ciclu determinat de vârfurile `1`, `2`, `3`, `4` şi `6`. Un arbore poate fi "agățat" de un vârf (care se numeşte **rădăcină**) astfel încât să-i atârne "ramurile" în jos. aşa cum este cazul lui `H` din aceeaşi figură. Un graf (neorientat) şi fără cicluri se numeşte **`pădure`**.

Componentele conexe ale unei păduri vor fi, fireşte, arbori! De observat că un arbore are `n-1` muchii, unde `n` este numărul său de vârfuri. Despre arbori vom mai discuta şi în capitolul următor. Acolo ne vom referi la arbori aranjați precum `H`, dar cu anumite orientări ale arcelor dintre vârfuri (aşa numitele *arborescențe*).



Metoda generală de rezolvare a problemei arborelui parțial de cost minim
Să considerăm o problemă economică ce conduce la problema arborelui parțial de cost minim:
Pentru construirea unei rețele interne de comunicație între secțiile unei întreprinderi s-a întocmit un proiect în care au fost trecute toate legăturile ce se pot realiza între secțiile întreprinderi. În vederea definitivării proiectului şi întocmirea necesarului de materiale etc., se cere să se determine un sistem de legături ce trebuie construit, astfel încât orice secție a întreprinderii să fie racordată la această rețea de comunicație, iar cheltuielile de construcție să fie minime.

După modelarea matematică necesară, o astfel de problemă se reduce la o problemă clasică de teoria grafurilor: *Date* $G=(V,E)$ *un graf şi* $C:E\to R$, $C(e)$*=costul muchiei* e*, să se determine un arbore parțial al lui* G, T^* *astfel încât* $c(T^*)=\min\{c(T)\}$; T-arbore parțial al lui G}, *unde* $C(T)=C(e_1)+C(e_2)+\ldots+C(e_{n-1})$. *(arborele parțial va avea* $n-1$ *muchii)*

Rezolvarea se face în $n-1$ paşi, numerotați de la 0 la $n-2$.

- pas 0:
  - se consideră $T^0=\{T_1^0,\ldots,T_n^0\}$ familie de arbori disjuncți, cu $T_i^0=(\{i\},\varnothing)$;
- pas k ($k = 0$, ..., $n-2$).
  - se dispune de $T^k=(T_1^k,\ldots,T_{n-k}^k)$ familia celor $n-k$ arbori disjuncți anterior construiți.

- se construieşte T^{k+1} astfel:
  - ⇒ se alege $T_s^k \in T^k$;
  - ⇒ dintre toate muchiile lui G, cu o extremitate în T_s^k şi cealaltă în $V \setminus V(T_s^k)$ se alege una de cost minim $e^*=v_s v_{j^*}$, unde $v_{j^*} \in V(T_{j^*}^k)$, $j^* \neq s$;
  - ⇒ $T^{k+1} := (T^k \setminus \{T_s^k, T_{j^*}^k\}) \cup T$, unde T este arborele obținut din T_s^k şi $T_{j^*}^k$, la care se adaugă muchia e^*.

Familia T^{n-1}, construită în pasul $n-2$ este formată dintr-un singur arbore: T_1^{n-1}, care este arborele parțial minimal.
În funcție de modul de alegere a arborelui T_s^k, avem doi algoritmi:

- **algoritmul lui Prim**: se alege întotdeauna arborele cu cele mai multe vârfuri dintre arborii familiei curente; aşadar, la fiecare pas se adaugă câte o muchie aceluiaşi arbore, cu o extremitate în el şi cealaltă în restul grafului;
- **algoritmul lui Kruskal**: se va alege unul din cei doi arbori cu proprietatea că sunt "uniți" printr-o muchie cu extremitățile în arbori diferiți; această alegere se realizează simplu prin sortarea muchiilor grafului crescător în raport cu costurile.

Algoritmul lui Prim
În implementarea acestui algoritm se foloseşte un vector suplimentar `Vecin`. În fiecare pas al algoritmului vom avea grijă ca acest vector să aibă următoarea semnificație:

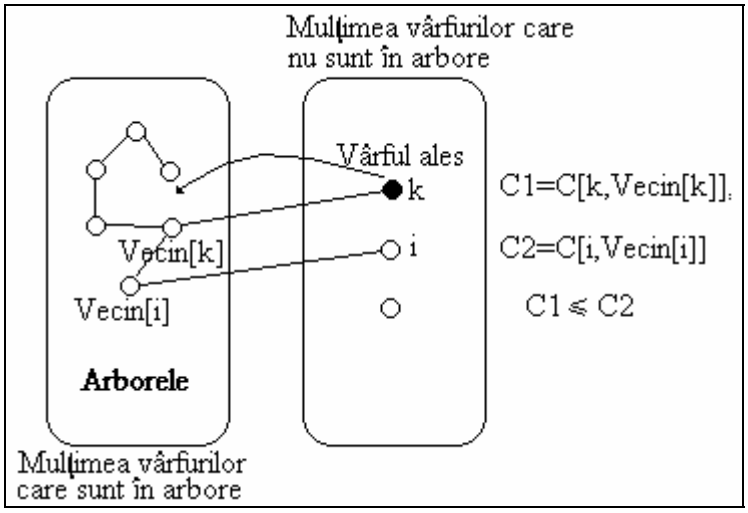
- dacă `Vecin[i]=0`, atunci înseamnă că `i` este în arborele parțial construit până în acel moment;
- dacă `Vecin[i]≠0`, atunci înseamnă că vârful `i` nu este în arbore, iar printre muchiile `(i,v)`, cu `v` din arbore, `(i,v0)` are costul cel mai mic.

Algoritmul lui Prim este:

- La început, se pune în arbore vârful `1` şi se consideră că acesta este vecinul tuturor vârfurilor din afara arborelui:
  `Vecin[1]:=0; Vecin[i]:=1, ∀i=2,...,n.` **(A)**
- La fiecare din ceilalți `n-1` paşi, se execută următoarele:
  - se determină cea mai mică (în sensul costului) muchie cu un vârf în afara arborelui (`i`) şi celălalt vârf în arbore (acesta se va lua `Vecin[i]`, pentru a avea cea mai mică muchie dintre cele ce le formează `i` cu vâfuri din arborele deja format): fie această muchie: `(k, Vecin[k])`; **(B)**
  - se adaugă muchia `(k,Vecin[k])` la arbore şi se actualizează costul; **(C)**
  - se actualizează vectorul `Vecin`, în conformitate cu definiția sa: **(D)**
    - ⇒ se pune `Vecin[k]=0`;
    - ⇒ pentru celălalte vârfuri `i` rămase în afara arborelui se verifică dacă muchia `(i,k)` nu este mai mică decât muchia veche `(i,Vecin[i])` (care lega pe `i` de arbore), în caz afirmativ punându-se `Vecin[i]=k`.

```
program Prim;
const max=10; infinit=10000;
var C: array[1..max, 1..max] of Integer; n,i,j,k,pas: Integer;
    Cost, min: Integer; Vecin: array[1..max] of Integer;
begin
  Write('n='); ReadLn(n);
  for i:=1 to n-1 do
      begin
        for j:=i+1 to n do
            begin
              Write('C[',i,',',j,']='); ReadLn(C[i,j]);
              if C[i,j]=0 then C[i,j]:=infinit;
              C[j,i]:=C[i,j]
```

```
            end;
          C[i,i]:=0
        end;
  for i:=1 to n do Vecin[i]:=1;                { A }
  Vecin[1]:=0;
  for pas:=1 to n do
      begin
        min:=infinit;                          { B }
        for i:=1 to n do
            if Vecin[i]<>0 then
               if C[i,Vecin[i]]<min then
                  begin
                    min:=C[i,Vecin[i]]; k:=i
                  end;
        WriteLn(k,'--',Vecin[k]);              { C }
        Cost:=Cost+C[k,Vecin[k]];
        Vecin[k]:=0;                           { D }
        for i:=1 to n do
            if Vecin[i]<>0 then
               if C[i,Vecin[i]] > C[i,k] then
                  Vecin[i]:=k
      end;
  WriteLn('Cost = ',Cost); ReadLn
end.
```



Algoritmul lui Kruskal

În implementarea **algoritmului lui Kruskal** se foloseşte metoda "*greedy*".

- se alege întâi muchia de cost minim,
- iar apoi se adaugă repetat (de `n-1` ori) muchia de cost minim nealeasă anterior şi care nu formează cu precedentele un ciclu.

Pentru a soluționa problema apariției unui ciclu prin adăugarea unei muchii, pentru fiecare vârf vom memora vârful său "tată", adică rădăcina arborelui din care face parte respectivul vârf, folosind un vector `Tata`.

Rădăcinile arborelui vor avea `Tata[i]=` **-**numărul_de_vârfuri_din_arborele_respectiv, deci se va pleca cu `n` arbori, fiecare format din câte un nod, care este şi rădăcina sa: `Tata[i] = -1`, la început **(A)**.

În momentul când muchia a `i` - a `(u,v)` devine candidată (muchiile se iau în ordinea crescătoare a costurilor lor (**B**)), se verifică cărui arbore (componentă) aparțin `u` şi `v` (**C**). Fie acestea `arb1`, respectiv `arb2` (**D**). Dacă `arb1` şi `arb2` sunt acelaşi arbore, atunci adăugarea muchiei `(u,v)` ar duce la apariția unui ciclu în acest arbore; dacă `arb1≠arb2`, atunci putem adăuga această muchie (**E**), după care va trebui să reunim cei doi arbori (**F**). La reunire, arborele mai mic se va lega de arborele mai mare, deci rădăcina ("tatăl") arborelui mai mare va deveni rădăcina şi pentru celălalt arbore, mai mic (**G**).

```
program Kruskal;
uses Crt;
const max=10;
var Muchie: array[1..max,1..3] of Integer;
    m,n,i,Cost: Integer; arb1, arb2: Integer;
    Tata: array[1..max] of Integer;
function Componenta(virf: Integer): Integer;    { C
}
begin
     while Tata[virf] > 0 do virf:=Tata[virf];
     Componenta:=virf
end;

procedure Reuneste(a,b: Integer);  { G }
var suma: Integer;
begin
     suma:=Tata[a]+Tata[b];
     if Tata[a]<Tata[b] then
        begin Tata[a]:=suma; Tata[b]:=a end
     else begin Tata[b]:=suma; Tata[a]:=b end
end;
begin
  ClrScr; Write('n='); ReadLn(n); Write('m='); ReadLn(m);
  WriteLn('Dati muchiile in ordine crescatoare a costurilor');
  { B }
  for i:=1 to m do
      begin
        WriteLn('Muchia nr. ',i); Write('u = '); ReadLn(Muchie[i,1]);
        Write('v = '); ReadLn(Muchie[i,2]); Write('cost = ');
        ReadLn(Muchie[i,3])
      end;
  for i:=1 to n do Tata[i]:=-1;   { A }
  Tata[1]:=0;
  Cost:=0;
  for i:=1 to m do
      begin
        arb1:=Componenta(Muchie[i,1]);{ D }
        arb2:=Componenta(Muchie[i,2]);
        if arb1<>arb2 then
           begin
             Reuneste(arb1,arb2); { F }
             Cost:=Cost+Muchie[i,3];
             WriteLn(Muchie[i,1],'--',Muchie[i,2]) {E}
           end
      end;
  WriteLn('Cost = ',Cost); ReadLn
end.
```

5. Exerciții recapitulative

Tehnicile de elaborare a algoritmilor prezentate în lecția anterioară (6) au fost folosite cu succes în cadrul algoritmilor referitori la grafuri din secțiunile acestei lecții. Există, însă, multe

situații când problema care se pune nu este formulată ca o problemă de grafuri, dar se poate reduce la o asemenea problemă. În cele ce urmează vă propunem trei probleme de acest gen. Depinde de perspicacitatea şi, totodată, de experiența rezolvitorului, să găsească cel mai eficient mod de a transpune în relații matematice o problemă.

1. Circuit hamiltonian. Se dâ harta rutieră a unei țări. Se cere să se determine toate posibilitățile de a efectua o excursie prin toate oraşele de pe hartă, trecând prin fiecare oraş exact o singură dată şi întorcându-ne în oraşul de plecare.
2. Colorarea vârfurilor unui graf. Se dă harta administrativă a unei țări, în care sunt puse în evidență județele (în număr de `n`). Se pune problema colorării județelor, astfel încât, dacă două județe sunt vecine, ele să aibe culori diferite. Sunt disponibile `m` culori distincte.
3. Închiderea tranzitivă a unei relații binare. Pe un lac sunt aşezate `n` plăci de lemn, de formă dreptunghiulară (date prin coordonatele colțurilor stânga-sus şi dreapta-jos), care plutesc deasupra apei şi care, eventual, pot acoperi suprafețe comune de apă. Pe o placă se aşază o pisică, iar pe alta un şoarece. Poate ajunge pisica la şoarece, mergând doar pe plăci, fără a păşi prin apă? Dacă da, să se enumere toate plăcile pe care poate să ajungă pisica, iar dacă nu să se precizeze dacă există o suprafață acoperită de toate plăcile pe care poate ajunge pisica.

Lecția 8
STRUCTURI DINAMICE DE DATE

1. Tipul referință

Am văzut în lecția anterioară că un graf se poate memora sub forma matricei sale de adiacență. Dacă, însă, graful are foarte multe noduri, folosirea matricei de adiacență este ineficientă. De asemenea, dacă vrem să facem un program pentru un concurs real de admitere în liceu, nu vom putea lucra cu vectori, deoarece nu putem declara vectori de lungimi apropiate de situații reale (500 de candidați, de pildă). Astfel, ar trebui să dispunem de o structură nouă de date, un fel de *listă*, care să păstreze mai multe informații înlănțuite între ele, într-o zonă de memorie mai largă.
Sunt, de asemenea, unele cazuri în care avem nevoie de structuri speciale de date, de exemplu, pentru a memora o ierarhie oarecare, ca de pildă un arbore genealogic. Pentru asemenea cazuri şi nu doar, structurile de date prezentate până acum, numite *statice*, nu ne ajută prea mult. Toate inconvenientele pot dispare, însă, odată cu utilizarea structurilor *dinamice* de date.

Variabile statice, variabile dinamice

Variabilele pot fi *statice* sau *dinamice*. Cele **statice** sunt alocate în timpul compilării, în zone bine definite de memorie. Structura, tipul şi locul lor din memorie nu se pot modifica în timpul execuției programului. Statice sunt toate variabilele folosite până acum.

Limbajul Pascal oferă posibilitatea alocării memoriei în timpul executării programului, pentru unele variabile, numite *dinamice*, în funcție de necesitate şi, de asemenea, există posibilitatea eliberării memoriei ocupate de ele. Variabilele **dinamice** trebuie să aibă tipul bine definit, însă nu se vor declara în secțiunea "`var`". De asemenea, accesul la astfel de variabile nu se face direct. Lor li se pune în corespondență un tip **referință,** în mod biunivoc, despre care se spune că referă sau indică ("pointează").

Variabila de tip referință poate conține referiri numai la variabila dinamică care i-a fost pusă în coresponență. Referirea se realizează prin memorarea în variabila de tip referință a adresei unde este stocată variabila dinamică. Corespondența între variabila dinamică şi tipul de referință permite cunoaşterea structurii variabilei dinamice.

Pentru variabilele de tip referință se va aloca, în timpul compilării, un spațiu de memorie de 4 octeți, care va conține adresa de memorie a variabilei dinamice referite.

Variabilele dinamice se alocă într-o zonă de memorie numită **heap**, diferită de zona fixă a programului, unde se memorează variabilele statice.

O variabilă dinamică referită va ocupa un spațiu de memorie corespunzător tipului ei: 2 octeți pentru `Integer`, 6 octeți pentru `Real`, 10 octeți pentru un şir de 9 caractere (`String[9]`) etc..

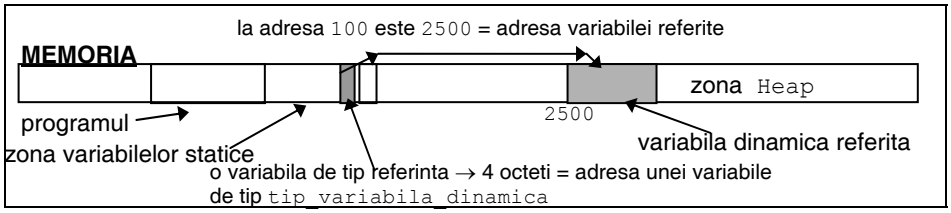
Definirea unui tip referință

Definirea unui tip referință se poate face în secțiunea `type` astfel:

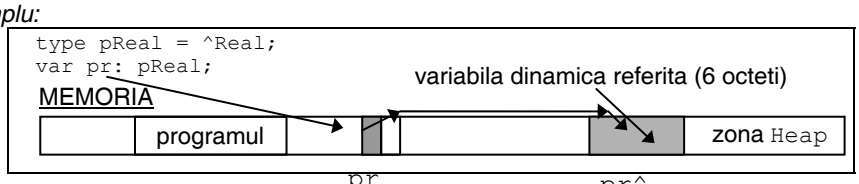
```
type tip_referintă = ^tip_variabilă_dinamică;
```

(se citeşte "pointer la...").

Mulțimea valorilor de tip `tip_referintă` constă într-un număr nelimitat de adrese. Fiecare adresă identifică o variabilă de tip `tip_variabilă_dinamică`. La această mulțime de valori se mai adaugă o valoare specială, numită **`nil`**, care nu identifică nici o variabilă.



Exemplu:

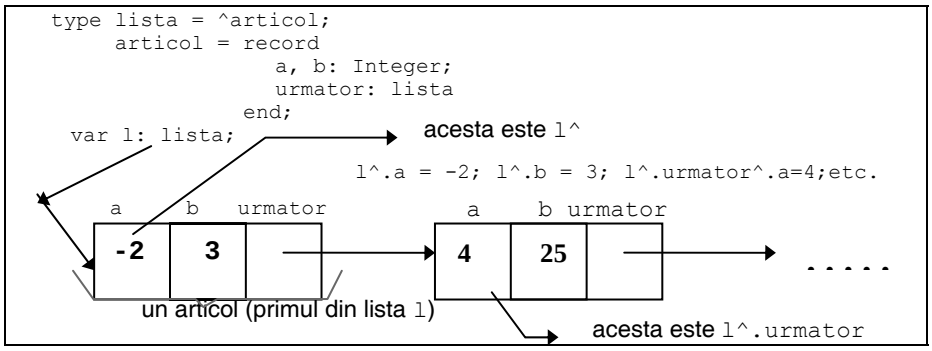


Este permis ca, în momentul întâlnirii tipului variabilei dinamice, acesta să nu fie cunoscut încă (referire înainte); acest tip trebuie declarat mai târziu, în aceeaşi declarație de tip.

De exemplu:

```
type p_complex = ^complex;
     complex = record re, im: Real end;
var r1, r2: p_complex; r3: ^complex; r4: ^Char;
```

O altă facilitate a limbajului constă în posibilitatea utilizării tipurilor care se autoreferă (sunt definite recursiv):



Utilizarea variabilelor dinamice. Avantaje

♦ Memorarea unei variabile dinamice se realizează în două faze:

- alocarea zonei de memorie pentru variabila dinamică, cu procedura **New**;
- memorarea efectivă, adică depunerea, la adresa pregătită, a datelor corespunzătoare variabilei dinamice.

♦ Eliberarea zonei de memorie corespunzătoare unei variabile dinamice, ocupată cu procedura New, se realizează cu procedura **Dispose**.

Exemplu:

```
program TestPointer;
type complex = record re, im: Real end; pComplex = ^complex;
var pz: pComplex;
begin
  New(pz); pz^.re:=1.2; pz^.im:=5.6;
  WriteLn('Nr. complex: ',pz^.re, ' ', pz^.im); Dispose(pz); ReadLn
end.
```

Cu variabilele dinamice se pot face face toate operațiile care se pot executa cu datele de respectivul tip.

Printre avantajele utilizării pointerilor se numără:

* folosirea unui spațiu de memorie redus, în cazul utilizării unor structuri complexe de date;
* folosirea mai eficientă a spațiului de memorie, prin eventuala reutilizare a sa (după `Dispose`).

Exemplu:

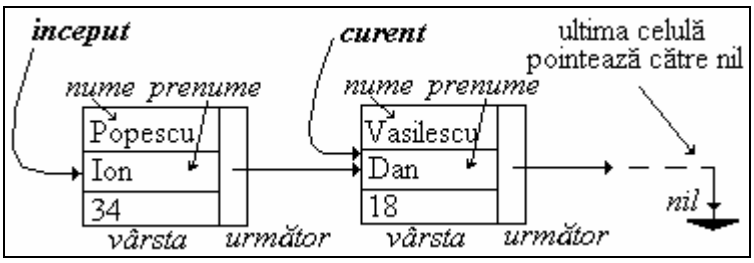
```
type persoana = record
                  nume, prenume: String; {2*256 octeti = 512 octeti}
                  virsta: Integer {2 octeti, deci 514 octeti, in total}
                end;
     pPersoana = ^persoana; { 4 octeti }
     mult_pers_1 = array[1..50] of persoana;
     mul_pers_2 = array[1..50] of pPersoana;
var M1: mult_pers_1; {50*514=25700 octeti}
    M2: mult_pers_2  {50*4=200 octeti, restul e in heap !}
```

Pentru a înlătura dezavantajul dimensiunii foarte mari a vectorilor (50 de elemente), vom realiza o listă înlănțuită, în care fiecare articol (nod) al listei va conține informațiile despre o persoană:

```
type pPersoana = ^persoana;
     persoana = record
                  nume, prenume: String[20];
                  virsta: Integer; urmator: pPersoana
                end;
var inceput, curent: pPersoana;
    {elementul de la inceput si cel curent din lista}
```

Exemplu: Crearea unei liste de persoane şi căutarea unei persoane în listă.

```
program CuListaDinamica;
type pPersoana = ^persoana;
     persoana = record
                  nume, prenume: String[20];
                  virsta: Integer; urmator: pPersoana
                end;
var n,p: String[20]; v: Integer; gasit: Boolean; raspuns: Char;
    inceput, sfirsit: pPersoana;
begin
  inceput:=Nil;
  repeat
    Write('Nume='); ReadLn(n);
    Write('Prenume='); ReadLn(p); Write('Virsta='); ReadLn(v);
    New(curent);
    with curent^ do begin nume:=n; prenume:=p; virsta:=v end;
    inceput:=curent;
    Write('Continuati [D/N] ? '); ReadLn(raspuns)
  until raspuns in ['n','N'];
  Write('Nume cautat = '); ReadLn(n);
  curent:=inceput; gasit:=False;
  while (curent<>Nil) and (not gasit) do
    if curent^.nume=n then gasit:=True else curent:=curent^.urmator;
  if gasit then
     WriteLn('Prenume=',curent^.prenume,', virsta=',curent^.virsta)
  else WriteLn('Persoana inxexistenta in lista...'); ReadLn
end.
```



2. Stive şi cozi

Listele, implementate dinamic, sunt foarte utile atunci când se lucrează cu multe informații, pe care vectorii se dovedesc incapabili a le stoca, sau ineficienți.
Un caz particular de liste (simplu înlănțuite) îl constituie *stivele* şi *cozile*. Acestea implementează două mecanisme diferite de intrare şi ieşire a elementelor din listă. La ambele feluri de liste, un nod al listei este o înregistrare ce conține o informație (`info`), precum şi un pointer (indicator) către următorul element al listei (`prec` sau `urm`).

Prezentare generală

- **Stiva** este o structură dinamică de date reprezentată de o listă simplu înlănțuită în care mecanismul de intrare - ieşire a elementelor este de tip `LIFO` - ultimul intrat este primul ieşit (*last in - first out*). Astfel, vom memora o stivă în felul următor:

```
type Stiva = ^Celula;
     Celula = record
                info: Integer; { informatia }
                prec: Stiva  { precedentul element din stiva }
              end;
var S: Stiva;
```

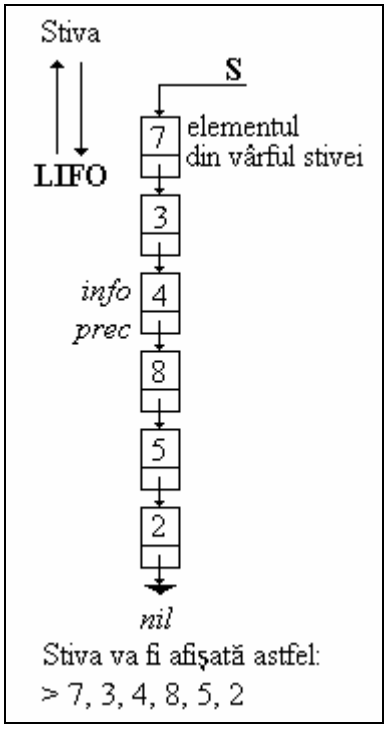
Aşadar, este de ajuns un pointer către primul element al stivei, pentru a realiza atât operația de adăugare a unui element (numită `Push`), cât şi cea de eliminare (`Pop`), deoarece ambele operații se realizează prin partea superioară a listei.

- **Coada** este o structură dinamică de date reprezentată de o listă simplu înlănțuită în care mecanismul de intrare - ieşire a elementelor este de tip `FIFO` - primul intrat este primul ieşit (*first in - first out*). Astfel, vom memora o coadă în felul următor:

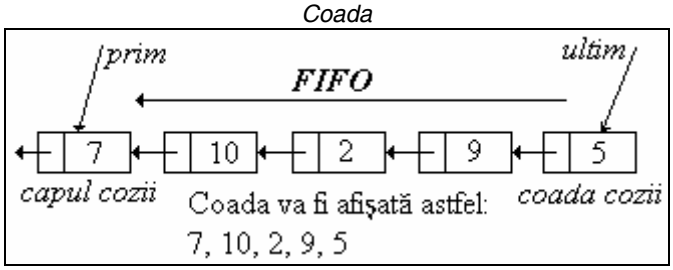
```
type  PCelula = ^ Celula;
      Celula = record
                 info: Integer; { informatia }
                 urm: PCelula { urmatorul la coada }
               end;
      Coada = record prim, ultim: Pcelula end;
var C: Coada;
```

Aşadar, în cazul cozii, avem nevoie de doi pointeri, unul către primul element al cozii (*capul cozii*), iar altul către ultimul său element (*coada cozii*), deoarece introducerea în listă se face prin spate, iar eliminarea prin față. (Există şi o variantă de coadă circulară, în care elementele sunt legate într-un cerc, iar cei doi pointeri, indicând capul şi coada cozii, sunt undeva pe acest cerc.).

Stiva



Coada



Programe demonstrative

Prezentăm două programe demonstrative care utilizează aceste structuri de date. Elementele din listă sunt numere întregi. Introducerea unui element se face prin simpla scriere a sa, dar trebuie ca elementul să nu fie nici `0`, nici `-1`. Eliminarea unui element (conform mecanismului implementat (`LIFO` sau `FIFO`) se realizează prin introducerea lui `-1`. Oprirea programului se face introducând `0`.

1. Programul cu stive. Prin apelul procedurii `Init(S)` se inițializează o stivă `S`, prin `Push(S,elem)` se adaugă elementul întreg `elem`, iar prin `Pop(S,elem)` se scoate `elem` din stiva `S`.

```
program CuStiva;
type Stiva = ^Celula;
     Celula = record
                info: Integer;
```

```
               prec: Stiva
             end;
procedure Init(var S: Stiva);
begin S := Nil end;
procedure Push(var S: Stiva; elem:
Integer); { pune }
var C: Stiva;
begin
  New(C); C^.info := elem;
  C^.prec := S; S := C
end;
procedure Pop(var S: Stiva; var
elem: Integer); { scoate }
var C: Stiva;
begin
  if S = Nil then
     begin
       Write('Stiva goala ...');
       elem := -1
     end
  else
     begin
       C:=S; elem:=C^.info;
       S:=C^.prec; Dispose(C)
     end
end;
procedure Afis(S: Stiva);
var C: Stiva;
begin
  C := S;
  Write('Stiva este: ',#16);
  while C <> Nil do
        begin
          Write(C^.info,',');
          C := C^.prec
        end;
  WriteLn
end;
var S: Stiva; elem: Integer;
begin
  Init(S); Afis(S);
  repeat
    Write('Dati elementul (-1 =
scoate, 0 = stop): ');
    ReadLn(elem);
    if elem <> 0 then
     if elem <> -1 then
        begin
          Push(S,elem);
          Afis(S)
        end
     else
      begin
        Pop(S,elem);
        WriteLn('Am scos ', elem);
        Afis(S)
      end
  until elem = 0;
end.
```

2. Programul cu coada. Acesta funcționează la fel ca şi cel anterior, doar că procedurile de adăugare şi eliminare se numesc `Pune`, respectiv `Scoate`, iar mecanismul este `FIFO`.

```
program CuCoada;
type PCelula = ^Celula;
     Celula = record
                info: Integer;
                urm: PCelula
              end;
     Coada = record
               prim,ultim:PCelula
             end;
procedure Init(var C: Coada);
begin
  C.prim:=Nil; C.ultim:=Nil
end;
procedure Pune(var C: Coada; elem:
Integer);
var P: PCelula;
begin
  New(P); P^.info:=elem;
  P^.urm:=Nil;
  if C.prim=nil then
     {nici un element}
     begin
       C.prim:=P; C.ultim:=P
     end
  else
     begin
       C.ultim^.urm:=P;C.ultim:=P
     end
end;
procedure Scoate(var C: Coada; var
elem: Integer);
var P: PCelula;
begin
 P := C.prim;
 if C.prim = Nil then
  begin
    Write('Coada este vida ...');
    elem := -1
  end
 else
  begin
    elem := C.prim^.info;
    C.prim := C.prim^.urm;
    Dispose(P)
  end
end;
procedure Afis(C: Coada);
var P: PCelula;
begin
  Write('Coada este: ');
  P := C.prim;
  while P <> Nil do
        begin
          Write(P^.info,',');
          P := P^.urm
        end;
  WriteLn
end;
var C: Coada; elem: Integer;
begin
  Init(C);
  Afis(C);
  repeat
    Write('Dati elementul (-1 =
scoate, 0 = stop): ');
    ReadLn(elem);
    if elem <> 0 then
      if elem <> -1 then
        begin
          Pune(C,elem);
          Afis(C)
        end
      else
        begin
          Scoate(C,elem);
          WriteLn('Am scos ',
elem);
          Afis(C)
        end
  until elem = 0;
end.
```

3. Liste dublu înlănțuite

În cazul listelor dublu înlănțuite avem, spre deosebire de stive şi cozi, următoarele noi elemente:

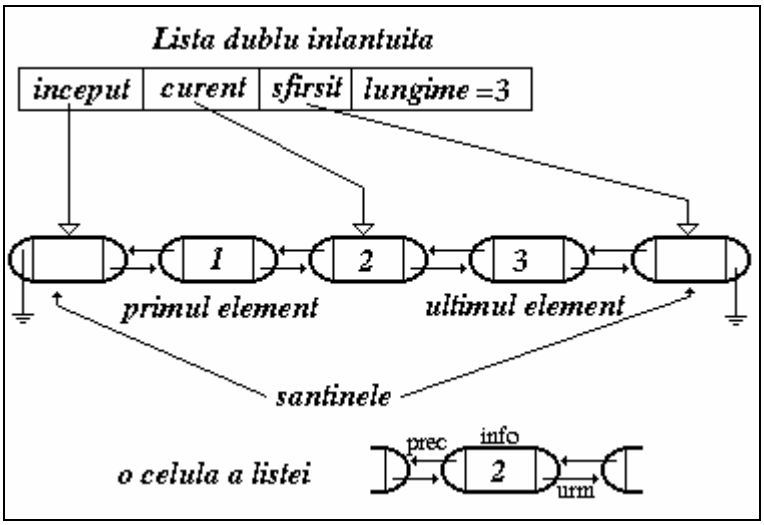
- există doi pointeri speciali: `inceput` şi `sfirsit` care point-ează către două celule extreme ale listei, dar care *nu* fac parte din listă; ei se numesc *santinele*;
- există un pointer numit `curent` care indică întotdeauna elementul curent din listă;
- fiecare element a listei este legată prin doi pointeri (`prec` şi `urm`) de elementele dinaintea şi de după el din cadrul listei;
- avem un câmp `lungime`, care va indica lungimea listei.

Astfel, lista de numere `1`, `2`, `3`, `4`, `5` va fi memorată ca în figura de mai jos.

Operațiile ce se cer a se efectua cu o astfel de listă sunt:

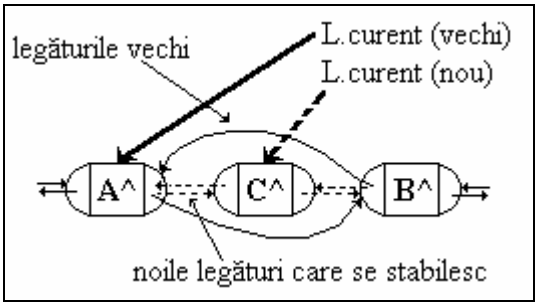
- inițializarea listei;
- adăugarea unui element la sfârşitul listei;
- inserarea unui element înaintea elementului curent din listă;
- ştergerea elementului curent din listă.
- afişarea listei.

Când se adaugă sau se inserează un nou element în listă, acel element devine cel curent. Când elementul curent se şterge din listă, locul său este preluat de elementul care îl succeda, în cadrul listei.



În momentul în care se introduce sau se elimină un element din listă, dispar unele legături şi apar altele. De pildă, să presupunem că avem o listă `L` şi, înaintea elementului curent se inserează o informație `elem`. Atunci vom creea o nouă celulă, cu numele `C`, în care vom pune această informație. Să numim celula curentă `A`, iar cea de după ea `B`. Atunci, va trebui să rupem legăturile dintre `A` şi `B`, stabilind, de asemenea, legături între `A` şi `C` şi între `C` şi `B`. Totodată, lungimea listei va creşte cu o unitate. Aceste lucruri sunt evidențiate în procedura de mai jos şi în figura asociată.

```
procedure Insereaza(var L: Lista; elem: tip_info);
var A,B,C: PCelula;
begin
   New(C); A:=L.curent; B:=L.curent^.urm; C^.info:=elem; C^.prec:=A;
   C^.urm:=B; A^.urm:=C; B^.prec:=C; L.lung:=L.lung+1; L.curent:=C
end;
```



Procese aproape inverse se realizează la eliminarea elementul curent din listă. Fireşte, acest lucru se poate realiza doar dacă lista nu este vidă. Pentru a elibera efectiv zona de memorie ocupată de elementul eliminat, se apelează procedura `Dispose`. Eliminarea elementului curent va presupune legarea elementului ce-l precede cu cel ce îl succede.

```
procedure Sterge(var L: Lista);
var C: PCelula;
begin
  with L do
    if lung>0 then
       begin
         C:=curent; C^.prec^.urm:=C^.urm;
         C^.urm^.prec:=C^.prec; curent:=C^.urm; Dispose(C);
         lung:=lung-1
       end
end;
```

Succesorul elementului eliminat devine element curent. De asemenea, lungimea listei scade cu o unitate.

În continuare prezentăm o nouă variantă a programului pentru admiterea în liceu. De această dată, candidații sunt puşi într-o listă dinamică, dublu înlănțuită. Astfel, numărul de candidați va fi limitat doar de memoria rămasă disponibilă în calculator, deci poate fi foarte mare, comparativ cu cazul memorării elevilor într-un vector.

Pentru a realiza ordonarea elevilor după un anumit criteriu (alfabetic sau după medii, descrescător), am implementat o variantă a algoritmului de sortare "*bubble-sort*", care foloseşte o listă `L` în locul unui vector. În esență, algoritmul este acelaşi. Deosebiri apar în parcurgerea listei, precum şi în cazul elementelor care se compară: în loc de `x[i]` şi `x[i+1]`, acum se compară elementele `L.curent^.info` şi `L.curent^.urm^.info`, de fapt acele câmpuri ale lor care sunt cheie în ordonarea efectuată. În program, secvența care implementează algoritmul "*bubble-sort*" este încadrată.

```
{$X+}
program CuListe;
uses Crt;
type elev = record nume: String; nota1, nota2, media: Real end;
     tip_info=elev;
     PCelula = ^Celula;
     Celula = record info: tip_info; prec, urm: PCelula end;
     Lista = record inceput, curent,  sfirsit: PCelula; lung: Integer end;
procedure Init(var L: Lista);
begin
  with L do
     begin
       lung:=0; inceput:=nil; sfirsit:=nil; inceput^.urm:=sfirsit;
       inceput^.prec:=nil; sfirsit^.prec:=inceput;
       sfirsit^.urm:=nil; curent:=inceput
```

```
     end
end;
procedure PozitioneazaLaInceput(var L: Lista);
begin
  while not (L.curent=L.inceput^.urm) do L.curent:=L.curent^.prec
end;
procedure Adauga(var L: Lista; elem: tip_info);
var C: PCelula;
begin
  New(C);
  with C^ do
    begin
      info:=elem; prec:=L.sfirsit^.prec; urm:=L.sfirsit^.prec^.urm
      { sau L.sfirsit}
    end;
  with L do
    begin
      lung:=lung+1; sfirsit^.prec^.urm:=C;
      sfirsit^.prec:=C; curent:=C
    end
end;
procedure Sterge(var L: Lista);
var C: PCelula;
begin
  with L do
    if lung>0 then
       begin
         C:=curent; C^.prec^.urm:=C^.urm; C^.urm^.prec:=C^.prec;
         curent:=C^.urm; Dispose(C); lung:=lung-1
       end
end;
procedure Insereaza(var L: Lista; elem: tip_info);
var A,B,C: PCelula;
begin
  New(C); A:=L.curent; B:=L.curent^.urm; C^.info:=elem; C^.prec:=A;
  C^.urm:=B; A^.urm:=C; B^.prec:=C; L.lung:=L.lung+1; L.curent:=C
end;
procedure Afis(L: Lista);
var i: Integer;
begin
  PozitioneazaLaInceput(L); i:=1;
  while not (L.curent = L.sfirsit) do
    begin
      with L.curent^.info do
           WriteLn(i,'. ',nume:20, nota1:6:2,nota2:6:2, media:6:2);
      L.curent:=L.curent^.urm; i:=i+1
    end;
  WriteLn('Total : ',i-1,' elevi.')
end;
procedure AdaugareElev(L: Lista);
var E: elev;
begin
  WriteLn('Adaugare elev');
  with E do begin
      Write('Dati numele : '); ReadLn(nume); Write('Dati notele : ');
      ReadLn(nota1,nota2); media:=(nota1+nota2)/2 end;
  Adauga(L,E)
end;
procedure EliminareElev(var L: Lista);
var numele: String; gasit: Boolean;
begin
  WriteLn('Eliminare elev'); PozitioneazaLaInceput(L);
```

```
  WriteLn('Dati numele elevului : '); ReadLn(numele); gasit:=False;
  while (not (L.curent = L.sfirsit)) and (not gasit) do
    if L.curent^.info.nume = numele then gasit:=True
    else L.curent:=L.curent^.urm;
  if gasit then Sterge(L) else WriteLn('Nu exista acest elev...')
end;
procedure CautareElev(L: Lista);
var numele: String; gasit: Boolean;
begin
  WriteLn('Cautare elev'); PozitioneazaLaInceput(L);
  WriteLn('Dati numele elevului : '); ReadLn(numele); gasit:=False;
  while (not (L.curent = L.sfirsit)) and (not gasit) do
    if L.curent^.info.nume = numele then gasit:=True
    else L.curent:=L.curent^.urm;
  if gasit then
     with L.curent^.info do WriteLn(nume,' ',nota1:6:2,nota2:6:2,media:6:2)
  else WriteLn('Elev inexistent...')
end;
procedure ListareEleviDupaNume(L: Lista);
var gata: Boolean; aux: elev;
begin
  WriteLn('Listare elevi dupa nume'); {ordonare elevi dupa nume:bubble-sort}
  repeat
    gata:=True; {ma pozitionez la inceput si parcurg lista}
    PozitioneazaLaInceput(L);
    while not (L.curent^.urm=L.sfirsit) do
      begin
        if L.curent^.info.nume > L.curent^.urm^.info.nume then
           begin
             gata := False; aux := L.curent^.info;
             L.curent^.info:=L.curent^.urm^.info; L.curent^.urm^.info:=aux
           end;
        L.curent:=L.curent^.urm
     end
  until gata;
  Afis(L) { afisare lista }
end;
procedure ListareEleviDupaMedii(L: Lista);
var gata: Boolean; aux: elev;
begin
  WriteLn('Listare elevi dupa medii');{ordonare elevi:bubble-sort}
  repeat
    gata:=True; {ma pozitionez la inceput, parcurg lista}
    PozitioneazaLaInceput(L);
    while not (L.curent^.urm=L.sfirsit)do
      begin
        if L.curent^.info.media < L.curent^.urm^.info.media then
           begin
             gata := False; aux := L.curent^.info;
             L.curent^.info:=L.curent^.urm^.info; L.curent^.urm^.info:=aux
           end;
        L.curent:=L.curent^.urm
      end
  until gata;
  Afis(L)
end;
var L: Lista; optiune: Char;
begin
  Init(L);
  repeat
    ClrScr;WriteLn('Meniu:');WriteLn; WriteLn('1 = adaugare elev');
    WriteLn('2 = stergere elev'); WriteLn('3 = cautare elev');
```

```
    WriteLn('4 = listare dupa nume'); WriteLn('5 = listare dupa medii');
    WriteLn('0 = oprire program'); WriteLn; Write('Optiune=');
    ReadLn(optiune); ClrScr;
    case optiune of
         '1': AdaugareElev(L); '2': EliminareElev(L);
         '3': CautareElev(L); '4': ListareEleviDupaNume(L);
         '5': ListareEleviDupaMedii(L)
    end;
    if optiune in ['1'..'5'] then ReadKey
 until optiune='0'
end.
```

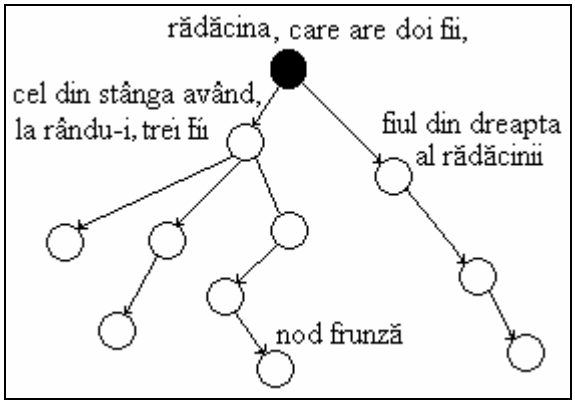
4. Arbori binari

Arborescențe în digrafuri

Fie $H=(V,E)$ un digraf (graf orientat). Se numeşte *rădăcină* a lui G un vârf $v_0 \in V$, astfel încât oricare ar fi un vârf $v \in V$, există cel puțin un drum de la v_0 la v. Dacă $H=(V,E)$ este un digraf, prin *graful suport* al lui H vom înțelege graful obținut din H, prin renunțarea la orientarea arcelor. H se numeşte **arborescență** dacă are o rădăcină şi graful său suport este arbore.

În informatică, arborescențele sunt numite, prin abuz de limbaj, **arbori**, specificându-se rădăcina şi considerând implicită orientarea muchiilor corespunzător parcurgerii drumului unic de la rădăcină la fiecare vârf. Fiecare vârf are astfel, nişte *fii*, adică vecinii imediat următori pe drumul de la rădăcină în jos (către *frunze*, adicâ noduri fără fii). Un arbore cu cel mult doi fii se mai numeşte şi **arbore binar**.

De exemplu:



Privit invers, un arbore seamănă, într-adevăr, cu un arbore din natură (un copac), astfel încât denumirea de *arbore*, ca şi cele de *frunză* sau *rădăcină* se justifică. Observăm că putem considera arborele ca fiind organizat pe mai multe *nivele*. Primul nivel este cel al rădăcinii. Urmează nivelul fiilor acesteia ş.a.m.d., până la ultimul nivel, cel al ultimelor frunze. În continuare ne vom ocupa, în mod deosebit, de arborii binari, deoarece, aşa cum vom arăta mai târziu, studiul arborilor oarecare se reduce la studiul arborilor binari.

Implementarea arborilor binari

Referirea unui arbore binar şi, implicit, definirea sa, va fi făcută printr-un pointer către nodul său rădăcină. Fiecare nod din arbore este o înregistrare cu următoarele elemente:

- o informație `info` de tip întreg;
- doi pointeri către cei doi fii (subarborii stâng şi drept) ai nodului: `stg` şi `dr`.

În programul care urmează, vom construi astfel de arbori, care au, în plus, următoarea proprietate: fiecare nod din arbore este mai mare sau egal cu nodurile din fiul stâng şi mai mic decât nodurile din fiul drept (din punct de vedere al câmpului `info`). Un astfel de arbore se numeşte **arbore de căutare - sortare**.

O *căutare* a unui element în astfel de arbori este, într-adevăr, uşor de realizat: dacă elementul căutat este identic cu informația din nod, atunci căutarea se încheie cu succes, dacă nu, atunci se pleacă pe una din cele două direcții: dacă elementul căutat este mau mic, atunci se merge pe fiul stâng, altfel pe fiul drept. Dacă se încearcă o trecere dincolo de un nod terminal, deci la `nil`, atunci căutarea eşuează. Procedura din următorul program demonstrativ returnează subarborele având ca rădăcină elementul găsit în arborele în care s-a căutat.

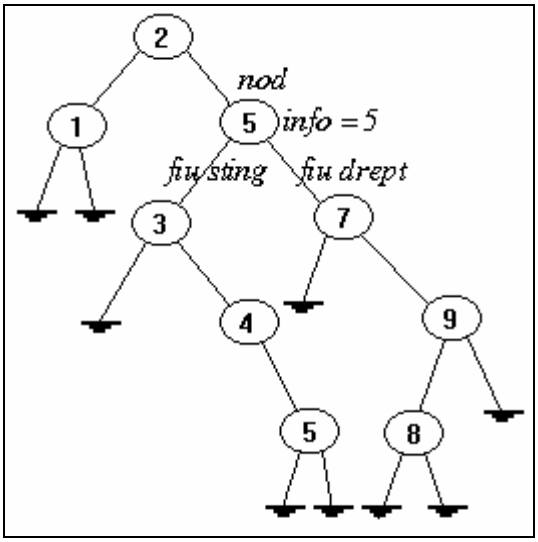
Un arbore binar poate fi parcurs în trei feluri:

- în *inordine*: se parcurge mai întâi, recursiv, în inordine, fiul stâng, apoi rădăcina, apoi fiul drept;
- în *postordine*: se parcurge fiul stâng, apoi cel drept, în final rădăcina;
- în *preordine*: se parcurge rădăcina, fiul stâng şi apoi fiul drept.

Se observă că parcurgerea în inordine a unui astfel de arbore duce la afişarea în ordine crescătoare a elementelor din nodurile arborelui, motiv pentru care astfel de arbori pot fi considerați şi de *sortare*.

Dacă se inserează, pe rând, elementele următoare în arbore: `2`, `5`, `7`, `3`, `4`, `9`, `1`, `5`, `8`, `0=stop`, atunci se obține arborele din figura următoare.

Programul care succede figura implementează structura de arbore discutată, precum şi principalele operații realizabile cu astfel de structuri de date.




```
program ArboriBinari;
type arbore = ^nod;
     nod = record info: Integer; stg, dr: arbore end;
procedure Init(var A: arbore);
begin A := Nil end;
{$S-}
```

```
procedure Inserare(var A: arbore; elem: Integer);
var UnNod: nod;
begin
  if A <> Nil then
   if elem <= A^.info then Inserare(A^.stg, elem) else Inserare(A^.dr, elem)
  else
   begin
     New(A); UnNod.info:=elem; UnNod.stg:=Nil; UnNod.dr:=Nil; A^:=UnNod
   end
end;
```

```
procedure Cautare(A: arbore; elem: Integer; var nodul: arbore);
begin
 if A = Nil then nodul := Nil
 else
   if A^.info = elem then nodul := A
   else
     if elem <= A^.info then Cautare(A^.stg, elem, nodul)
     else Cautare(A^.dr, elem, nodul)
end;
```

```
procedure PreOrdine(A: arbore);
begin
  if A <> Nil then
     begin Write(A^.info,','); PreOrdine(A^.stg); PreOrdine(A^.dr) end
end;
procedure PostOrdine(A: arbore);
begin
  if A <> Nil then
     begin
       PostOrdine(A^.stg); PostOrdine(A^.dr); Write(A^.info,',')
     end
```

```
end;
procedure InOrdine(A: arbore);
begin
  if A <> Nil then
     begin Inordine(A^.stg); Write(A^.Info,','); InOrdine(A^.dr) end
end;
{$S+}
var A, B: arbore; elem: Integer;
begin
  Init(A);
  repeat
    Write('Dati elementul de inserat (0 = stop): '); ReadLn(elem);
    if elem <> 0 then Inserare(A,elem)
  until elem = 0;
  WriteLn('Arborele in preordine : '); PreOrdine(A); WriteLn;
  WriteLn('Arborele in postordine : '); PostOrdine(A); WriteLn;
  WriteLn('Arborele in inordine', ' => elementele sortate: ');
  InOrdine(A); WriteLn;
  repeat
    Write('Dati elementul cautat'); Write(' (0 = stop): '); ReadLn(elem);
    if elem <> 0 then
       begin
         Cautare(A, elem, B);
         if B <> Nil then
            begin
              WriteLn('L-am gasit,', ' arborele in inordine este: ');
              InOrdine(B); WriteLn
            end
         else WriteLn('Nu l-am gasit.')
       end
  until elem = 0;
end.
```

5. Aplicație: derivare formală

Pentru a exemplifica un mod de utilizare a arborilor binari, să reamintim că orice expresie aritmetică poate fi memorată sub forma unui arbore binar. Prin urmare, dacă avem o expresie aritmetică oarecare, putem obține din ea arborele binar asociat ei. Pentru a deriva (formal) respectiva expresie, vom deriva arborele asociat ei, obținând arborele expresiei derivate. Din acest arbore vom construi expresia derivată.

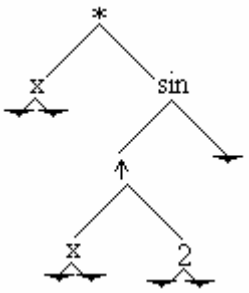
Derivarea formală este o problemă clasică: dându-se o funcție `f(x)=E(x)`, în care `E(x)` este o expresie în funcție de `x`, conținând operatorii `+,-,*,/,^`, precum şi funcțiile trigonometrice, logaritmică şi exponențială, se cere să se obțină `f'(x)=D(x)`, în care `D(x)` este expresia obținută din `E(x)` prin derivare formală, adică aplicând regulile fundamentale de derivare:

`(f*g)'=f'*g+f*g', (f(g(x))'=f'(g(x))*g'(x), (sin(x))'=cos(x)` etc..

Pentru o expresie algebrică, numim **formă poloneză inversă** notația postfixată a expresiei, adică cea în care mai întâi apar operanzii, apoi operatorul.

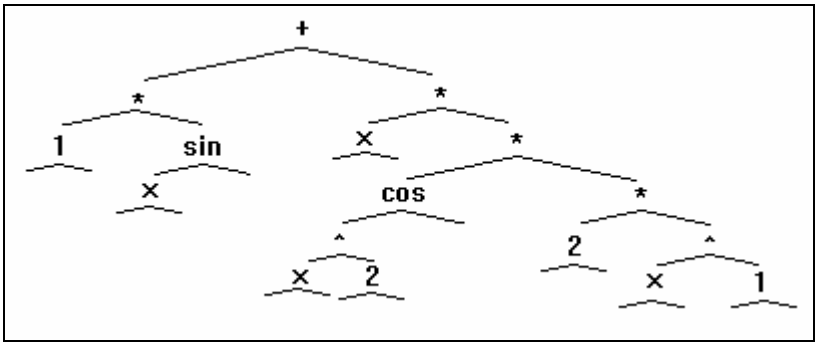
Astfel, o expresie va fi reprezentată prin arborele binar corespunzător formei poloneze inverse

Exemplu: pt. expresia x*sin(x^2) vom avea arborele:



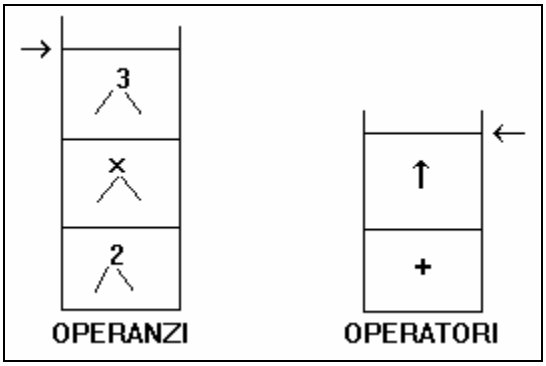
- funcțiile unare, precum `sin` aici, vor avea fiul din dreapta `nil`, iar numerele şi `x` vor avea ambii fii `nil`;
- forma poloneză inversă asociată acestui arbore este: `x x 2 ^ sin *`.

Derivarea acestei expresii ar însemna construirea următorului arbore şi apoi scrierea expresiei corespunzătoare, cu operatorii binari în forma infixă, iar funcțiile (unare) în forma prefixă (în figura de mai jos nu am mai figurat `nil`-urile):



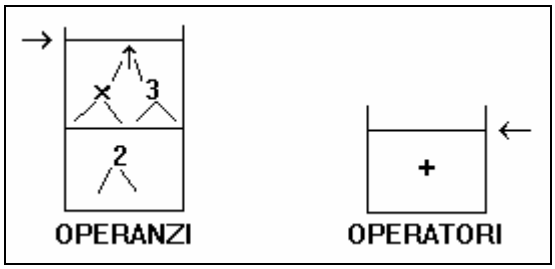
Fireşte, transformarea dintr-un arbore într-altul s-ar baza pe transformări simple de arbori, care ar reflecta derivările primitive: `f+g→f'+g'`, `f*g→(f'+g)*(f+g')` etc..

Vom utiliza o structură arborescentă binară clasică. Cum vom transforma, însă, o expresie obişnuită într-un arbore? Ideea este asemănătoare celei de la evaluatorul de expresii aritmetice: acolo se folosesc doua stive, una a operatorilor şi funcțiilor - văzute ca operatori unari - , cealaltă a operanzilor, care sunt numere.
Aici vom folosi tot două stive, una a operatorilăor şi funcțiilor, cealaltă a operanzilor, care, de data aceasta, vor fi arbori mai mici, din care se vor forma arbori mai mari, folosind drept rădăcină ceea ce se afla în vârful stivei cu operații la respectivul moment.
Vom introduce în stiva operațiilor arborele micuț: $/^{2}\backslash$ (şi nu numărul `2`), apoi în stiva operațiilor `+`, apoi arborele $/^{x}\backslash$ în stiva operanzilor.
Observăm că `^` (ridicarea la putere) este de prioritate mai mare ca `+`, deci nu vom face `2 + x`, ci îl vom introduce şi pe `^` în stiva operațiilor, apoi pe $/^{3}\backslash$ în cea a operanzilor.

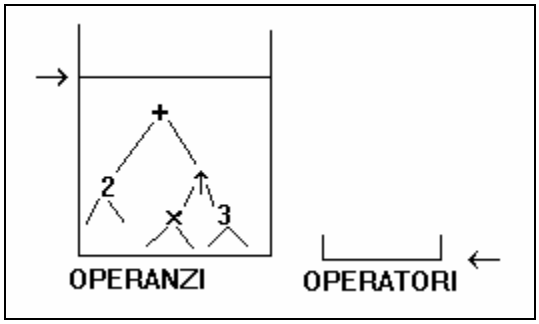


În acest moment va trebui să scoatem cei doi arbori mici din stivă şi, împreună cu ^ (scos din stiva operațiilor) drept rădacină, să formăm un nou arbore:

pe care îl vom introduce în stiva operanzilor:



Singura operație rămasă este + şi, procedând ca mai înainte, obținem:



În acest moment, stiva operatorilor este goală, iar stiva operanzilor conține exact arborele ce ne trebuie pentru a putea continua cu derivarea expresiei. Probleme deosebite apar atunci cind în vârful stivei operațiilor este - sau /, iar operația succesoare (în expresie) este tot -, respectiv /. (Aceste simboluri reprezintâ operații necomutative.)

Programul următor realizează aceste transformări şi derivări, însa el poate fi îmbunătățit de către cititor prin adăugarea de operații de simplificare a arborelui rezultat, corespunzător derivării. Aceste operații s-ar baza pe primitivele de genul:

f+0→→f , f*1→→f etc..

Un element de noutate care poate fi observat este prezența unor instrucțiuni grafice, mai ales în cadrul procedurii recursive de afişare a unui arbore binar. Utilizarea graficii are rolul de a face mai plăcută afişarea unui arbore binar; instrucțiunile grafice trebuie luate ca atare, ele nu influențează algoritmul cu nimic. Asupra procedurilor grafice apărute aici se va reveni în lecția 11. Singura observație care trebuie făcută aici este că, pentru a porni programul, trebuie cunoscută calea DOS către fişierul EGAVGA.BGI, din mediul Turbo-Pascal. Pe cuprinsul acestei cărți noi am considerat că aceasta este "C:\BP\BGI". În caz că nu este aşa, cititorul va înlocui acest şir de caractere cu cel corect, în procedura de inițializare grafică OpenGraph (în apelul procedurii standard InitGraph). (De, asemenea, precizăm că Line(x1,y1,x2,y2) trasează, pe display-ul grafic, o linie între punctele de coordonate (x1,y1) şi (x2,y2), unde coordonatele variază între 0 şi 639 (de la stânga la dreapta), respectiv între 0 şi 479 (sau 349) de sus în jos. OutTextXY(x,y,sir) afişează un şir de caractere în poziția specificată, iar CloseGraph închide modul grafic. ClearViewPort - în cazul de față - şterge ecranul grafic. Utilizarea subprogramelor grafice se declară prin uses Graph, iar în opțiunile de compilare din meniul mediului Turbo-Pascal trebuie să aveți la "Directories" calea către fişierul GRAPH.TPU.)

```
program Derivare;
uses Graph,Crt;
const lmax = 3; vmax = 50;
type expresie  =  string; informatie  =  string[lmax];
     vector  =  record
                  info:array[1..vmax] of informatie; nr:1..vmax
                end;
     arbore  =  ^nod;
     nod  =  record info:informatie; stg,dr:arbore end;
var expresia, expr_fpo, expr_der_fpo: expresie; vectorul: vector;
    arborele, arborele_derivat: arbore;
procedure OpenGraph;
var gd,gm: Integer;
begin
  gd:= Detect; InitGraph(gd,gm,'C:\BP\BGI'); {calea fisierului EGAVGA.BGI}
end;
procedure Vectorizeaza(expr: expresie; var vec: vector);
    procedure Schimba(var x, y: informatie);
    var a: informatie;
    begin a := x; x := y; y := a end;
var i,j,k: Integer;
begin
  i := 0; k := 0;
  while i<Length(expr) do
    begin
      Inc(i);
      if expr[i] in ['s','c','l','e'] then
         begin
           Inc(k);
           if expr[i]<>'l' then vec.info[k]:=expr[i]+expr[i+1]+expr[i+2]
           else vec.info[k] := expr[i]+expr[i+1]+' ';
           if expr[i] = 'l' then Inc(i)
           else
             begin vec.info[k] := vec.info[k]+expr[i+2]; Inc(i,2) end
         end
      else
        if expr[i] in ['(',')','+','-','*','/','^','x'] then
           begin Inc(k); vec.info[k] := expr[i] end
        else
         begin { cifre };
```

```
          Inc(k); vec.info[k] := '        '; j := 1;
          while
            not (expr[i] in ['(',')','+','-','*','/','^','s','c','l','e'])
          do begin vec.info[k][j] := expr[i]; Inc(i); Inc(j) end;
          Dec(i)
        end
    end;
  vec.nr := k
end;
procedure Arborizeaza(vec: vector; var arb: arbore);
const topmax = 25;
var i,top1,top2: Integer;
    operator: array[1..topmax] of String[lmax];
    operand: array[1..topmax] of arbore;
  function prioritate(op:char):byte;
  begin
    case op of
         '(',')':prioritate := 0; '+','-':prioritate := 1;
         '*','/':prioritate := 2; '^':prioritate := 3;
         's','c','l','e':prioritate := 4
    end
  end;
begin { arborizeaza }
  i := 0; top1 := 0; top2 := 1; operator[top2] := '(';
  while (i<=vec.nr) and (top2>0) do
    begin { 1 }
      Inc(i);
      if vec.info[i][1] in ['x','0'..'9'] then
         begin
           Inc(top1); New(arb); arb^.info := vec.info[i];
           arb^.stg := nil; arb^.dr := nil; operand[top1] := arb;
         end
      else
        if vec.info[i][1]='(' then begin Inc(top2);operator[top2]:='(' end
        else
         begin { 2 }
           while (top2>0) and (not (operator[top2][1] in ['(',')'])) and
                 (prioritate(operator[top2][1])>=prioritate(vec.info[i][1]))
             do begin
                  if operator[top2][1] in ['l','c','s','e'] then
                     begin
                       New(arb); arb^.info := operator[top2];
                       arb^.stg := operand[top1]; arb^.dr := nil;
                       operand[top1] := arb
                     end
                  else begin { + - * / ^ }
                         New(arb); arb^.info := operator[top2];
                         arb^.stg:=operand[top1-1];arb^.dr:=operand[top1];
                         operand[top1-1]:=arb; Dec(top1)
                       end;
                Dec(top2)
              end; { while - 2}
          if top2>0 then
             if (operator[top2] <> '(') or (vec.info[i][1] <> ')') then
                begin Inc(top2); operator[top2] := vec.info[i] end
             else Dec(top2)
      end; { else ...}
  end; { while }
  if (i=vec.nr) and (top2=0) then begin New(arb); arb := operand[1] end
  else begin New(arb); arb^.info:='old'; arb^.stg:=nil; arb^.dr:=nil end
end;
{$S-}
```

```
procedure Deriveaza(arbi: arbore; var arb: arbore);
var arb1, arb2, arb3, arb4, arb5, arb6, arb7 : arbore;
begin
  New(arb); arb^.info := 'new'; arb^.stg := nil; arb^.dr := nil;
  case arbi^.info[1] of
    '0'..'9': begin arb^.info := '0' end;
    'x': begin arb^.info := '1' end;
    '+','-': begin
               Deriveaza(arbi^.stg,arb3); arb^.stg := arb3;
               Deriveaza(arbi^.dr,arb4); arb^.dr := arb4;
               arb^.info := '+'
             end;
    '*': begin
           New(arb1); New(arb2); Deriveaza(arbi^.stg,arb3);
           Deriveaza(arbi^.dr,arb4);
           arb^.info:='+'; arb1^.stg:=arb3; arb1^.dr:=arbi^.dr;
           arb1^.info:='*'; arb2^.stg:=arbi^.stg; arb2^.dr:=arb4;
           arb2^.info:='*'; arb^.stg:=arb1; arb^.dr:=arb2;
           end;
    '/': begin
           New(arb1); New(arb2); New(arb5);
           Deriveaza(arbi^.stg,arb3); Deriveaza(arbi^.dr,arb4);
           arb1^.info:='*'; arb1^.stg:=arbi^.stg; arb1^.dr:=arb4;
           arb2^.info:='*'; arb2^.stg:=arb3; arb2^.dr:=arbi^.dr;
           arb5^.info:='-'; arb5^.stg:=arb1; arb5^.dr:=arb2;
           New(arb6); New(arb7);
           arb6^.info:='2'; arb6^.stg:=nil; arb6^.dr:=nil;
           arb7^.info:='^'; arb7^.stg:=arbi^.dr; arb7^.dr:=arb6;
           arb^.info:='/'; arb^.stg:=arb5; arb^.dr:=arb7
         end;
    's': begin
           New(arb1); arb1^.info := 'cos';
           arb1^.stg:=arbi^.stg; arb1^.dr:=nil; Deriveaza(arbi^.stg,arb2);
           arb^.info := '*'; arb^.stg := arb1; arb^.dr := arb2
         end;
    'c': begin
           Deriveaza(arbi^.stg,arb4); New(arb1); New(arb2); New(arb3);
           arb1^.info:='0'; arb1^.stg:=nil; arb1^.dr:=nil;
           arb2^.info:='sin'; arb2^.stg:=arbi^.stg; arb2^.dr:=nil;
           arb3^.info:='*'; arb3^.stg:=arb2; arb3^.dr:=arb4;
           arb^.info:='-'; arb^.stg:=arb1; arb^.dr:=arb3
         end;
    'l': begin
           Deriveaza(arbi^.stg,arb1);
           arb^.info:='/'; arb^.stg:=arb1; arb^.dr:=arbi^.stg
         end;
    'e': begin
           New(arb2); arb2^.info:=arbi^.info;
           arb2^.stg:=arbi^.stg; arb2^.dr:=arbi^.dr;
           Deriveaza(arbi^.stg,arb1);
           arb^.info:='*'; arb^.stg:=arb2; arb^.dr:=arb1
         end;
    '^': begin
           Deriveaza(arbi^.stg,arb1); New(arb5); New(arb2);
           arb5^.info:='1'; arb5^.stg:=nil; arb5^.dr:=nil;
           arb2^.info:='-'; arb2^.stg:=arbi^.dr; arb2^.dr:=arb5;
           New(arb3); arb3^.info:='^'; arb3^.stg:=arbi^.stg; arb3^.dr:=arb2;
           New(arb4); arb4^.info:='*'; arb4^.stg:=arbi^.dr; arb4^.dr:=arb3;
           arb^.info:='*'; arb^.stg:=arb4; arb^.dr:=arb1
         end
  end
end;
```

```
{$S+}
{$S-}
procedure Tipareste(arb: arbore; nivel,x0: Integer);
const dy=30;
var i, dx: Integer;
begin
  dx := GetMaxX; for i := 1 to nivel do dx := dx div 2;
  if arb=nil then OutTextXY(x0+dx, nivel*dy, #207)
  else begin
         OutTextXY(x0+dx, nivel*dy, arb^.info);
         Line(x0+dx, nivel*dy+5, x0+dx div 2, (nivel+1)*dy-5);
         Line(x0+dx, nivel*dy+5, x0+dx+dx div 2, (nivel+1)*dy-5);
         Tipareste(arb^.stg,nivel+1,x0);Tipareste(arb^.dr,nivel+1,x0+dx)
       end
end;
{$S+}
{$S-}
procedure FormaPolonezaInversa(arb: arbore; var expr: expresie);
var expr1, expr2: expresie;
begin
  expr := '';
  if arb<>nil then
     begin
       FormaPolonezaInversa(arb^.stg, expr1);
       FormaPolonezaInversa(arb^.dr, expr2);
       expr := expr1 + ' ' + expr2 + ' '+arb^.info
     end
end;
{$S+}
begin { programul principal }
  ClrScr; WriteLn(' Dati expresia care va fi derivata ! ');
  ReadLn(expresia); expresia := expresia+')';
  vectorizeaza(expresia,vectorul); New(arborele);
  Arborizeaza(vectorul,arborele);
  OpenGraph; SetTextJustify(CenterText, CenterText);
  OutTextXY(GetMaxX div 2, 10, Copy(expresia,1,Length(expresia)-1));
  FormaPolonezaInversa(arborele, expr_fpo);
  OutTextXY(GetMaxX div 2, GetMaxY-30, expr_fpo);
  Tipareste(arborele,1,0); Deriveaza(arborele, arborele_derivat);
  ReadLn; Dispose(arborele); ClearViewPort;
  FormaPolonezaInversa(arborele_derivat, expr_der_fpo);
  OutTextXY(GetMaxX div 2, GetMaxY-30, expr_der_fpo);
  Tipareste(arborele_derivat,1,0); ReadLn;
  Dispose(arborele_derivat); CloseGraph
end.
```

După cum se observă şi din corpul programului principal anterior, paşii algoritmului sunt:

* se citeşte expresia de derivat: `expresia`;
* se memorează expresia într-un vector (`vector`), cu care se va lucra în continuare;
* se construieşte arborele asociat expresiei, din acest vector;
* se derivează acest arbore, obținându-se arborele derivat al expresie; derivarea se face cu o procedură recursivă, în care se studiază toate cazurile de rădăcină a arborelui în cauză.

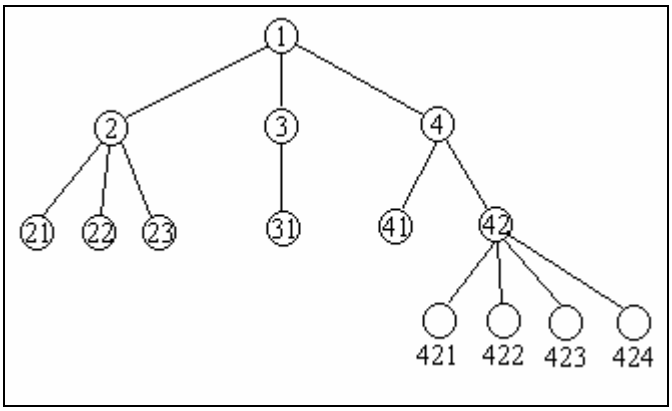
De fiecare dată se obține şi forma postfixată (forma poloneză inversă) a arborelui (derivat sau nu), apoi fiecare din cei doi arbori se afişează sub formă grafică. De observat marea asemănare între procedura `arborizeaza` şi funcția de evaluare a unei expresii algebrice, expusă în lecția 5, precum şi naturaleța procedurii recursive de derivare `deriveaza`.

Lăsăm ca exercițiu cititorului obținerea expresiei derivate în forma normală, pornind de la arborele derivat.

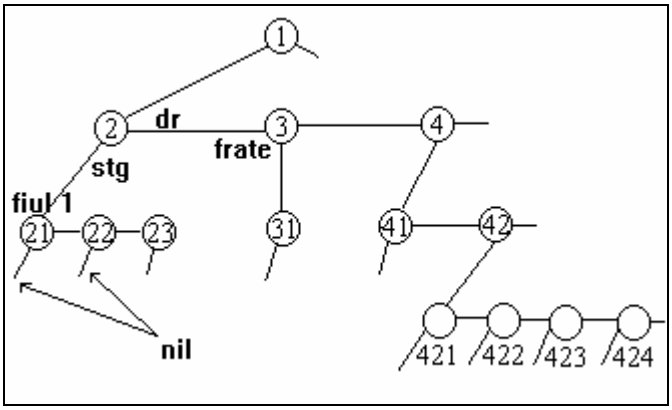
6. Memorarea arborilor oarecare în arbori binari

Prezentare teoreticâ

Într-adevăr, să considerăm un arbore oarecare, ca cel din figura următoare. Observăm că putem lega rădăcina `1` de nodul `2`, iar apoi, nodul `2` poate fi legat de primul fiu al său (`21`) şi de următorul fiu al rădăcinii, deci `3`, despre care se spune că este un *frate* a lui `2`. Procedând astfel pentru toate nodurile din arbore, vom obține un arbore binar, cu două legături: cea din stânga este către primul fiu, iar cea din dreapta către primul frate din dreapta al acestui fiu.



Astfel, arborele din figura anterioară se va memora în arborele binar de mai jos:



Arborii oarecare îi vom memora sub forma unei structuri ce cuprinde o informație `info` (de tip `Integer`), un număr de fii (`NrFii`), precum şi un vector de pointeri (referințe) către toți fii nodului în cauză (`fiu: array[1..max] of arbore`). În cele ce urmează vom prezenta un program demonstrativ care va face următoarele:

- va citi un arbore oarecare `A` de la tastatură;
- va afişa acest arbore, pe vericală, aşa cum este afişată structura de directoare `DOS`, în urma unei comenzi `TREE`;
- va memora arborele `A` sub forma unui arbore binar `A2` (de tip `arbore2`);
- va afişa, în mod asemănător, acest arbore `A2`.

În privința procedurilor care execută aceste operații ale programului, trebuie să precizăm următoarele:

♦ procedura `Citeste` preia de la tastatură datele despre un arbore oarecare `A` în felul următor: se citeşte informația dintr-un nod (cel rădăcină), apoi numărul de fii pe care îi are acest nod; pentru fiecare fiu se va apela recursiv procedura, citindu-se subarborii care au ca rădăcini respectivii fii;

♦ procedura `TiparireÎnPreordine` (precum cazul ei particular dat de procedura `TiparireÎnPreordine2`) foloseşte caractere grafice speciale, pentru a desena structura de arbore. Aceste caractere grafice speciale formează diferite *antete*, care se afişează înaintea unui nod. (Aceste antete pot fi spații sau antete de verticale, care semnifică că suntem pe un nivel inferior, sau chiar antete depinzând de numărul de ordine al nodului, ca fiu al tatălui său. Steluța simbolizează `nil`.);

♦ procedura `TransfArb` este cea mai importantă, ea realizând transformarea unui arbore oarecare `A` într-un arbore binar `A2`; procedura este recursivă şi se apelează pentru primul fiu al rădăcinii lui `A` (notat `fiu1`), precum şi pentru toți ceilalți fii (între `2` şi `NrFii`), care se înlănțuie la dreapta, în arborele `A2`.

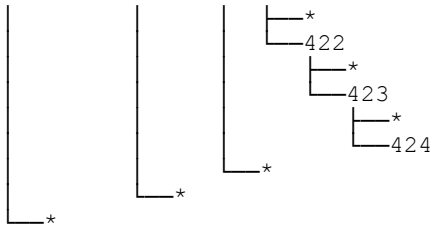
Implementarea algoritmului

```
program TransformArbOarecareInArbBinar;
uses Crt;
const max=10;
type arbore = ^nod;
     nod=record NrFii:Byte; info:Integer; fiu: array[1..10] of arbore; end;
     arbore2 = ^nod2;
     nod2 = record info: Integer; stg,dr: arbore2 end;
procedure TransfArb(a: arbore; var a2: arbore2);
var fiu1:arbore; c,b,d: arbore2; i: Byte;
begin
  if a<>nil then
     begin
       New(a2); a2^.info:=a^.info; a2^.stg:=nil; a2^.dr:=nil;
       if a^.NrFii>0 then
          begin
            fiu1:=a^.fiu[1]; TransfArb(fiu1,b); a2^.stg:=b; d:=b;
            for i:=2 to a^.NrFii do
                begin TransfArb(a^.fiu[i],c); b^.dr:=c; b:=b^.dr end
          end
     end
  else a2:=nil
end;
procedure TiparireInPreordine2(A: arbore2);
const stinga=#195#196#196; dreapta=#192#196#196; vertical=#179#32#32;
      MaxNivele=20;
var antet: array[1..MaxNivele] of String; k: Integer;
  procedure TPreordine(curent: arbore2);
  var i: Integer;
```

```
  begin
    for i:=0 to k do Write(antet[i]);
    if curent<>nil then WriteLn(curent^.info)
    else WriteLn('*');
    if curent<>nil then
       if (curent^.stg<>nil) or (curent^.dr<>nil) then
          if k<MaxNivele then
             begin
               if antet[k]=stinga then antet[k]:=vertical
               else antet[k]:='   ';
               k:=k+1; antet[k]:=stinga; TPreordine(curent^.stg);
               antet[k]:=dreapta; TPreordine(curent^.dr); k:=k-1
             end
  end;
begin k:=0; antet[k]:='-->'; TPreordine(A); end;
procedure Citeste(var A: arbore; nr, parinte: Integer);
var i: Byte;
begin
  New(A); Write('Dati info pt. fiul ',nr,' al lui ', parinte,': ');
  ReadLn(A^.info);
  if A^.info<>0 then
     begin
       Write('Dati numarul de fii', 'pentru nodul ',A^.info,' : ');
       ReadLn(A^.NrFii);
       for i:=1 to A^.NrFii do Citeste(A^.fiu[i],i,A^.info);
     end
  else A:=nil
end;
procedure TiparireInPreordine(A: arbore);
const stinga=#195#196#196; dreapta=#192#196#196; vertical=#179#32#32;
      MaxNivele=20;
var antet: array[0..MaxNivele] of String[3]; k: Integer;
  procedure TPreordine(curent: arbore);
  var j,i: Integer;
  begin
    for i:=0 to k do Write(antet[i]);
    if curent<>nil then WriteLn(curent^.info)
    else WriteLn('*');
    if curent<>nil then
       if curent^.NrFii>0 then
          if k<MaxNivele then
             begin
               if antet[k]=stinga then antet[k]:=vertical
               else antet[k]:='   ';
               k:=k+1;
               for j:=1 to curent^.NrFii-1 do
                   begin
                     antet[k]:=stinga;
                     TPreordine(curent^.fiu[j])
                   end;
               antet[k]:=dreapta;
               with curent^ do TPreordine(fiu[NrFii]);
               k:=k-1
             end
  end;
begin k:=0; antet[k]:='-->'; TPreordine(A); end;
var A: arbore; A2: arbore2;
begin
  ClrScr; Citeste(A,0,0); TiparireInPreordine(A);
  TransfArb(A,A2); TiparireInPreordine2(A2); ReadLn
end.
```

Iată un exemplu de funcționare a programului anterior pentru arborele oarecare considerat în figura dată în prezentarea teoretică:

```
Dati info pt. fiul 0 al lui 0: 1
Dati numarul de fii pentru nodul 1 : 3
Dati info pt. fiul 1 al lui 1: 2
Dati numarul de fii pentru nodul 2 : 3
Dati info pt. fiul 1 al lui 2: 21
Dati numarul de fii pentru nodul 21 : 0
Dati info pt. fiul 2 al lui 2: 22
Dati numarul de fii pentru nodul 22 : 0
Dati info pt. fiul 3 al lui 2: 23
Dati numarul de fii pentru nodul 23 : 0
Dati info pt. fiul 2 al lui 1: 3
Dati numarul de fii pentru nodul 3 : 1
Dati info pt. fiul 1 al lui 3: 31
Dati numarul de fii pentru nodul 31 : 0
Dati info pt. fiul 3 al lui 1: 4
Dati numarul de fii pentru nodul 4 : 2
Dati info pt. fiul 1 al lui 4: 41
Dati numarul de fii pentru nodul 41 : 0
Dati info pt. fiul 2 al lui 4: 42
Dati numarul de fii pentru nodul 42 : 4
Dati info pt. fiul 1 al lui 42: 421
Dati numarul de fii pentru nodul 421 : 0
Dati info pt. fiul 2 al lui 42: 422
Dati numarul de fii pentru nodul 422 : 0
Dati info pt. fiul 3 al lui 42: 423
Dati numarul de fii pentru nodul 423 : 0
Dati info pt. fiul 4 al lui 42: 424
Dati numarul de fii pentru nodul 424 : 0
-->1
    ├──2
    │   ├──21
    │   ├──22
    │   └──23
    ├──3
    │   └──31
    └──4
        ├──41
        └──42
            ├──421
            ├──422
            ├──423
            └──424
-->1
    ├──2
    │   ├──21
    │   │   ├──*
    │   │   └──22
    │   │       ├──*
    │   │       └──23
    │   └──3
    │       ├──31
    │       └──4
    │           ├──41
    │           │   ├──*
    │           │   └──42
    │           │       ├──421
```



7. Exerciții recapitulative

1. Se citesc numere întregi de la tastatură, până la întâlnirea numărului `0`. Se cere să se creeze două liste, una a numerelor negative, iar alta a numerelor pozitive prime.
2. Se dă o listă de numere întregi pozitive. Să se creeze o listă care să conțină doar numerele pare, apoi să se concateneze cele două liste.
3. Se dau doi arbori binari. Se cere să se înlocuiască fiecare nod al primului cu cel de al doilea arbore.
4. Se dă un arbore oarecare, informațiile din noduri fiind şiruri de caractere. Să se construiască o listă dublu înlănțuită care să conțină toate şirurile din nodurile arborilor, care au lungimile pare, apoi să se ordoneze această listă.

Lecția 9
FIŞIERE

1. Utilizarea fişierelor DOS

Odată ce lucrați cu Turbo-Pascal, sub sistemul de operare MS-DOS, noțiunea de *fişier* trebuie să vă fie familiară. Astfel, fişierele reprezintă principala formă de organizare a datelor pe disc. Atât programele noastre, texte scrise într-un limbaj sau altul, cât şi programele executabile obținute de compilatoare, precum şi acele colecții de date necesare funcționării acestor programe sunt fişiere. Ele sunt organizate în directoare, însă directoarele sunt privite tot ca fişiere, având un atribut suplimentar. Chiar şi eticheta de volum (de disc) este văzută ca un fişier, ba chiar şi ecranul, tastatura sau imprimanta. Există şi un fişier special, numit NUL, care reprezintă fişierul în care nu se poate scrie nimic şi de unde nu se poate citi nimic.
Identificatorul unui fişier cuprinde, în afara numelui propriu-zis, şi alte informații: calea de directoare până la el (inclusiv unitatea de disc pe care se află), câteva atribute referitoare la tipul fişierului (de sistem, ascuns, director etc.), extensia sa, precum şi data şi ora la care a fost creat.
Privind astfel un fişier, ca un element gestionat de sistemul de operare, contează mai puțin ce cuprinde sau cum poate fi prelucrat el (citit, scris, modificat, şters). Ceea ce importă sunt operațiile legate de atributele fişierului, de localizarea sa, lucruri de care se ocupă nişte proceduri şi funcții din **biblioteca DOS**, pentru care, ca şi Crt, atunci când este utilizată, va trebui să se precizeze explicit acest lucru: `uses DOS`.

Fiecărui fişier fizic extern (de pe disc) FISIER i se asociază o **variabilă de tip fişier**, internă, F cu ajutorul procedurii Assign, în felul următor: `Assign(F,FISIER)`. Declararea lui F ca variabilă fişier internă se face prin `var F: File`. În felul acesta, prin F se înțelege un fişier oarecare, dar vom vedea în paragraful următor că ne putem referi chiar numai la fişiere text sau numai la fişiere ce cuprind un anumit gen de date.

În cele ce urmează nu vom prezenta toate subprogramele din biblioteca DOS, ci doar câteva referitoare la fişiere, ce ne vor permite să realizăm un program de genul comenzii Tree a sistemului de operare DOS, care să ne afişeze întreaga structură arborescentă de directoare, cu rădăcina în directorul curent de pe unitatea curentă.

Ideea este simplă. Vom construi, ajutați fiind de programul din lecția anterioară, ultima secțiune, arborele asociat structurii de directoare respective, apoi vom afişa acest arbore în preordine, pe verticală. Nodurile arborelui vor fi numele de directoare. Pentru a vedea ce subdirectoare are un anumit director (în care ne aflăm la un moment dat), se folosesc două proceduri speciale FindFirst şi FindNext, care găsesc primul şi respectiv următorul fişier din directorul curent, care au un anumit format sau tip. Dintre acestea se selectează doar acelea a căror atribut conține identificatorul de director, cu excepția directoarelelor '`.`' şi '`..`' ce corespund directorului curent, respectiv directorului părinte. Subdirectoarele curente vor fi fii nodului ce corespunde directorului curent. Apoi, cu procedura ChDir vom schimba directorul, trecând din nodul curent într-unul din fii, pentru care se va construi, la fel, subarborele corespunzător (acest fiu va fi rădăcină în noul arbore). Ne întoarcem la tatăl subdirectorului (prin `ChDir('..')`), apoi trecem în următorul fiu, pentru care procedăm la fel, şi tot aşa, până se epuizează fii. Procedura recursivă care construieşte arborele cu rădăcina într-un director se numeşte Construieste. Ea se va apela din programul principal pentru directorul curent. La sfârşit, când arborele întreg este construit, acesta se

afişează cu procedura TiparireInPreordine, care la rândul său cuprinde o subprocedură recursivă TPreordine.
În cele ce urmează prezentăm programul şi comentăm diferite utilizări ale subprogramelor folosite din biblioteca DOS.

```
program Arb;
{ comanda "Tree" }
uses Dos;
const max=500;
type TipInfo = String[12]; { informatia din nodurile arborelui este
un nume de fisier, cu tot cu extensie }
     arbore = ^nod;
     nod = record
             info: TipInfo; NrFii: Byte;
             fiu: array[1..max] of arbore
           end;
procedure TiparireInPreordine(A: arbore);
const stinga=#195#196#196#196; dreapta=#192#196#196#196;
      vertical=#179#32#32#32; MaxNivele=20;
var antet: array[0..MaxNivele] of String[4]; k: Integer;
  procedure TPreordine(curent: arbore);
  var j,i: Integer;
  begin
    for i:=0 to k do Write(antet[i]);
    if curent<>nil then WriteLn(curent^.info)
    else WriteLn('*'); { nu se afisa niciodata }
    if curent<>nil then
       if curent^.NrFii>0 then
          if k<MaxNivele then
             begin
               if antet[k]=stinga then antet[k]:=vertical
               else antet[k]:=#32#32#32#32; { patru spatii }
               k:=k+1;
               for j:=1 to curent^.NrFii-1 do
                   begin
                     antet[k]:=stinga; { pentru primii fii }
                     TPreordine(curent^.fiu[j])
                   end;
               antet[k]:=dreapta; { ultimul fiu }
               TPreordine(curent^.fiu[curent^.NrFii]); k:=k-1
             end {TPreordine }
  end;
begin k:=0; antet[k]:='--> '; TPreordine(A); end;
```

Procedura Construieste va căuta, cu FindFirst şi FindNext toate fişierele (deci cu şablonul '*.*') FisDirector din directorul curent, care sunt directoare (au atributul Directory).
Directorul curent se află cu apelul GetDir(0,P), în care P este o variabilă de tip PathStr, adică un şir de caractere corespunzător căii curente, iar 0 este o constantă, asociată unei anumite unități de disc: 0 = unitatea curentă, 1 = A, 2 = B etc.. Din tot şirul P, reprezentând calea curentă, ne interesează numai numele directorului şi extensia sa (dacă directorul are extensie). Astfel, se desparte P în mai multe subşiruri prin: FSplit(P,D,N,E), unde D este calea până la directorul părinte inclusiv, iar N şi E sunt numele directorului şi extensia sa. Acestea din urmă se compun pentru a forma informația ce va fi ataşată nodului rădăcină al arborelui: A^.info:=N+E.

Odată determinat directorul curent, se caută toți fii săi prin secvența:

- determină primul fiu, dacă există:

```
FindFirst('*.*', Directory, FisDirector);
```

- cât timp există încă un fiu (fapt semnalat prin valoarea `0` în variabila de sistem `DosError` care e declarată în biblioteca `DOS`):
```
       while DosError = 0 do
             begin
```
- asignează numele fişierul fizic `FisDirector.Name` la variabila fişier `F` şi determină-i atributele (cu procedura `GetFAttr`):
```
              Assign(F, FisDirector.Name);
              GetFAttr(F,Attr);
```
- dacă printre aceste atribute se află şi cel de director (constanta `Directory`, cu valoare `16`, declarată în biblioteca (unit-ul) `DOS`), iar directorul nu este nici '`.`' şi nici '`..`', atunci adaugă fişierul director fiilor rădăcinii arborelui:
```
              if (Attr and Directory<>0) and (FisDirector.Name<>'.') and
                 (FisDirector.Name<>'..') then
                   begin
                     Inc(A^.NrFii); New(A^.fiu[A^.NrFii]);
                     A^.fiu[A^.NrFii]^.info:=FisDirector.Name
                   end;
```
- apoi treci la căutarea următorului fişier care se potriveşte şablonului dorit:
```
              FindNext(FisDirector)
            end;
```

În final, se va merge în fiecare dintre subdirectoarele fii şi se va proceda recursiv pentru continuarea construirii structurii arborescente.

Procedura completă şi programul principal sunt prezentate mai jos. (Se foloseşte un nou tip întreg de date: **`Word`**, predefinit astfel: `type Word=0..65535` (dubdomeniu al lui `LongInt`), deci se poate lucra cu numere întregi ce depăşesc `32767` (`MaxInt`), dar sunt doar pozitive.).

```
procedure Construieste(var A: arbore);
var FisDirector: SearchRec; F: File;
    Attr: Word; { adica intreg intre 0 si 65535 }
    P: PathStr; { String[80] } D: DirStr; { String[67] }
    N: NameStr; { String[8] } E: ExtStr; { String[4] }
    i: Integer;
begin
     A^.NrFii:=0; GetDir(0,P); FSplit(P,D,N,E); A^.info:=N+E;
     FindFirst('*.*', Directory, FisDirector);
     while DosError = 0 do
       begin
         Assign(F, FisDirector.Name); GetFAttr(F,Attr);
         if (Attr and Directory<>0) and (FisDirector.Name<>'.') and
            (FisDirector.Name <> '..') then
            begin
              Inc(A^.NrFii); New(A^.fiu[A^.NrFii]);
               A^.fiu[A^.NrFii]^.info:=FisDirector.Name
            end;
         FindNext(FisDirector)
       end;
     for i:=1 to A^.NrFii do
         begin
           ChDir(A^.fiu[i]^.info);
           Construieste(A^.fiu[i]); ChDir('..')
         end
end;
var A: arbore; Cale: PathStr;
begin
     New(A); GetDir(0,Cale); WriteLn('Arborele directorului ',Cale);
     Construieste(A); TiparireInPreordine(A); WriteLn
end.
```

În urma executării acestui program, se va afişa o structură arborescentă de genul:

```
Arborele directorului D:\MSOFFICE
--> MSOFFICE
    ├──SETUP
    │   └──MS-BTTNS
    ├──WINWORD
    │   ├──TEMPLATE
    │   ├──MACROS
    │   ├──STARTUP
    │   ├──LETTERS
    │   └──WORDCBT
    ├──EXCEL
    │   └──LIBRARY
    │       └──MSQUERY
    └──CLIPART
        └──PCSFILES
```

Ca exercițiu, propunem cititorului să realizeze un program similar în care să fie afişate şi fişierele obişnuite ale directoarelor, eventual cu litere mici şi după subdirectoarele fii.

2. Generalități despre variabilele fişier

În afara faptului că sunt necesare stocării programelor noastre pe suport magnetic de mare capacitate (disc), fişierele sunt necesare şi stocării datelor pe care le prelucrează aceste programe. Astfel, de pildă, dacă dorim să facem un program pentru concursul de admitere în liceu, va fi foarte necesar să stocăm pe disc datele introduse în memorie, pentru ca să le putem folosi şi altă dată. Astfel, va trebui să dispunem de un nou tip de date, asemănător tablourilor, cu un număr practic nelimitat de articole, însă, care să păstreze informațiile pe disc, nu în memoria internă a calculatorului. Acest tip de dată este fişierul, iar fişierele, în Turbo-Pascal, sunt de trei feluri: fişiere text (tipul de date `Text`), fişiere cu tip (`File of tip`) şi fişiere fără tip, aşa cum le-am utilizat în primul paragraf: `File`.

Indiferent de tipul unui fişier, vom asigna variabila de tip fişier `F` unui fişier extern, fizic, `FISIER`, prin apelul procedurii **`Assign`**, astfel: `Assign(F,FISIER)`. În continuare, vom lucra cu variabila `F`, dar, de fapt, ne vom referi, până la o eventuală reasignare a lui `F`, la fişierul propriu-zis `FISIER`, adică la fişierul de pe disc cu acest nume.

Pentru a înțelege mai bine utilizarea procedurii, să considerăm că avem în directorul curent un fişier cu numele '`TEST.TXT`', un fişier text. Vom declara `var F: Text`, după care vom scrie `Assign(F,'TEST.TXT')`. Orice operație asupra lui '`TEST.TXT`' va fi realizată folosind variabila `F`.

Operațiile ce se fac asupra unui fişier sunt de *citire* şi *scriere*, exact ca şi operațiile ce se pot face cu tastatura/ecranul, care sunt şi ele fişiere; un fişier poate fi asemănat cu un catalog de note al unei clase. Pentru a scrie şi citi, va trebui să deschidem fişierul. La sfârşit, când operațiile de citire/scriere s-au încheiat, fişierul va trebui să fie închis. Închiderea se realizează, indiferent de tipul fişierului, prin `Close(F)`, iar deschiderea depinde de tipul fişierului.

3. Fişiere text

Declarare. Operații

Variabilele de tip fişier text, ce se declară prin

```
var F: Text
```

corespund fişierelor fizice ce conțin texte oarecare, adică a acelor fişiere ce sunt văzute ca o succesiune de rânduri, fiecare rând fiind un şir de caractere (maxim 255 !).
Un fişier text `F` poate fi deschis pentru a citi din el cu **`Reset(F)`**, iar pentru a scrie la sfârşitul său cu **`Append(F)`**. Crearea sau rescrierea unui fişier text se realizează prin **`ReWrite(F)`**.

Operațiile de scriere şi citire sunt cele obişnuite pentru ecran şi tastatură. În fişierele text totul se scrie sub formă de şiruri de caractere, inclusiv valorile numerice sau cele booleene. Valorile booleene se scriu sub forma şirurilor de caractere '`TRUE` ' şi respectiv '`FALSE`'. Putem scrie valori ale unor expresii întregi, booleene, caracter, reale sau `String`, putem citi variabile de aceste tipuri, mai puțin `Boolean`.

- `procedure WriteLn(F: Text; e1, e2, ...);`
  (`F` este fişierul text, iar `e1`, `e2`, ... sunt expresii);
- `procedure ReadLn(F: Text; var v1, v2, ...);`
  (`F` este fişierul text, iar `v1`, `v2`, ... sunt variabile).

Liniile dintr-un fişier text se citesc (sau scriu) secvențial. Astfel, pentru a ajunge la cea de a zecea linie, se vor citi primele nouă. Putem să ne imaginăm existența unui *pointer (indicator) de fişier* care se poziționează, pe rând, la începutul liniilor citite.

Procedura de citire `Read` pentru fişiere text şi tastatură

În afara procedurii `ReadLn`, mai există şi procedura `Read`, care poate fi folosită pentru a citi atât dintr-un fişier text, cât şi de la tastatură. Deosebirea față de `ReadLn` constă în aceea că nu se ia în considerare sfârşitul de linie. Acesta trebuie citit separat, ca orice caracter. De fapt, sfârşitul de linie este urmat de o coborâre pe linia următoare. Avem de a face cu succesiunea de caractere `#13 #10`, adică `CR` (`Carriage Return`) şi `LF` (`Line Feed`), corespunzătoare apăsării tastei `Enter`.
Astfel, pot apare complicații atunci când se dorește citirea unui şir de caractere.
De pildă, dacă `x` este un număr real sau întreg, iar `s` este un şir de caractere, `Read(x)` sau `ReadLn(x)` au acelaşi efect, pe când pentru a face ceea ce face `ReadLn(s)` va trebui să scriem `Read(s)`; `ReadLn`, apelul lui `ReadLn` având drept scop citirea caracterelor de sfârşit de linie (`Enter`).
Dacă vrem să citim mai multe şiruri (de lungime necunoscută) de numere dintr-un fişier text. situate toate pe aceeaşi linie, vom scrie o secvență care să citească, până se ajunge la sfârşitul fişierului, fiecare linie în parte.

Funcțiile `EoLn` şi `EoF`

Faptul că am ajuns sau nu la sfârşitul unei linii ne este dat de funcția logică `EoLn`, care pentru un fişier text `F` se va apela prin `EoLn(F)`, iar pentru tastatură prin `EoLn`. Există şi o funcție logică `EoF` care ne indică faptul că am ajuns sau nu la sfârşitul fişierului. Se apelează prin `EoF(F)` pentru un fişier text `F`, respectiv prin `EoF` pentru tastatură.

```
program CitireCuRead;
var F: Text; x: Real;
begin
     Assign(F,'T.TXT'); Reset(F);
     while not EoF(F) do
           begin
             while not EoLn(f) do
                   begin Read(F,x); Write(x:6:2,', ') end;
             ReadLn(F); WriteLn; WriteLn('-------------------------')
           end;
     Close(F); ReadLn
end.
```

Pentru fişierul text 'T.TXT' de mai jos:

```
24.5 3.7 -4.76 9.80 -5.2
5.4 -7 8.10 4
```

rezultatul programului anterior va fi afişarea următoarelor:

```
 24.50,   3.70,  -4.76,   9.80,  -5.20,
-----------------------------------------
  5.40,  -7.00,   8.10,   4.00,
-----------------------------------------
```

Recomandăm utilizarea procedurii Read, atât pentru fişiere text, cât şi pentru ecran, în situații cu totul speciale, cum ar fi cele de genul anterior, în rest e mult mai comod de lucrat cu ReadLn.

Exemple de utilizare a fişierelor text

Următorul program citeşte de la tastatură câte trei numere reale, apoi le scrie într-un fişier text *'IN.TXT'*. Apoi, citind aceste trei numere, reprezentând laturile unui triunghi, scrie în fişierul text 'OUT.TXT' tipul triunghiului format.

```
program Triunghiuri;
var a,b,c: Real; F,G: Text; i: Byte;
begin
     Assign(F,'IN.TXT'); ReWrite(F);
     for i:=1 to 10 do
         begin
           Write('Dati a, b si c : '); ReadLn(a,b,c);
           WriteLn(F,a:5:2,' ',b:5:2,' ',c:5:2)
         end;
     Close(F); Reset(F); Assign(G,'OUT.TXT'); ReWrite(G);
     while not EoF(F) do
           begin
             ReadLn(F,a,b,c);
             if (a=b) and (b=c) then WriteLn(G,'echilateral')
             else if (a=b) or (b=c) then WriteLn(G,'isoscel')
                  else if (a*a=b*b+c*c) or (b*b=c*c+a*a) or
                          (c*c=a*a+b*b) then WriteLn(G,'dreptunghic')
                       else WriteLn(G,'oarecare')
           end;
     Close(F); Close(G)
end.
```

Un exemplu de rezultat al programului este dat mai jos. Sunt vizualizate conținuturile fişierelor 'IN.TXT' şi respectiv 'OUT.TXT':

```
2.00  2.00  5.00
3.00  4.00  5.00
2.00  4.00  3.00
7.00  7.00  7.00
5.00  4.00  3.00
8.00  8.00  9.00
9.00  8.00  9.00
4.00  5.00  6.00
4.00  6.00  8.00
8.00  8.00  8.00
```

```
isoscel
dreptunghic
oarecare
echilateral
dreptunghic
isoscel
oarecare
oarecare
oarecare
echilateral
```

S-a folosit şi funcția EoF, care indică faptul că s-a ajuns la sfârşit de fişier sau nu; "*while not EoF(f) do ...*" se traduce prin: "*cât timp încă n-ai ajuns la sfârşitul fişierului execută ...*"

Un alt exemplu de utilizare a fişierelor text este următorul program care concatenează două fişiere text într-unul singur (primul dintre ele):

```
program Concatenare;
var F,G: Text; linie, nume_unu, nume_doi: String;
```

```
begin
  Write('Dati numele primului fisier    : '); ReadLn(nume_unu);
  Write('Dati numele celui de al doilea : '); ReadLn(nume_doi);
  Assign(F,nume_unu); Append(F); Assign(G,nume_doi); Reset(G);
  while not eof(G) do begin ReadLn(G,linie); WriteLn(F,linie) end;
  Close(G); Close(F);
  WriteLn('Am pus totul in ',nume_unu,'.'); ReadLn
end.
```

Programul citeşte toate liniile din cel de al doilea fişier şi le adaugă la sfârşitul primului fişier. În acest scop, fişierul destinație (primul) va fi deschis cu `Append`.

Utilizarea parametrilor din linia de comandă

De multe ori, mai ales când se lucrează cu fişiere, vrem să trimitem programului nişte parametri în linia de comandă. (Acest lucru se realizează punând parametrii în cadrul opțiunii `Parameters...` din meniul `Run`, în mediul Turbo-Pascal.).

De pildă, vrem să realizăm o variantă a programului anterior, în care numele celor două fişiere ce se concatenează să se precizeze în linia de comandă. Astfel, dacă numele programului executabil anterior ar fi `CONCAT.EXE`, ne-ar place să pornim acest program cu o linie de comandă de genul: `concat unu.txt doi.txt`, în care spunem că fişierul `doi.txt` se adaugă fişierului `unu.txt`, sau măcar `concat unu.txt`, în care să ni se ceară numai numele celui de al doilea fişier.

În acest sens ne ajută funcțiile **`ParamCount`** care ne returnează numărul de parametri din linia de comandă (de pildă 2 pentru un apel `concat unu.txt doi.txt`), parametrii fiind despărțiți de spații, şi **`ParamStr`**, care ne dă chiar aceşt parametri: `ParamStr(1)` = numele primului parametru, `ParamStr(2)` = numele celui de al doilea, ş. a. m. d., iar numele programului pornit este `ParamStr(0)`.

```
program Concat; { concat.pas }
var F,G: Text; linie, nume_unu, nume_doi: String;
begin
  if ParamCount>=2 then
     begin
       WriteLn('Am preluat numele fisierelor din linia de comanda.');
       nume_unu:=ParamStr(1); nume_doi:=ParamStr(2)
     end
  else
    if ParamCount=1 then
       begin
         WriteLn('Am preluat numele primului fisier din linia de comanda.');
         nume_unu:=ParamStr(1);
         Write('Dati numele celui de al doilea : '); ReadLn(nume_doi)
       end
    else
       begin
         WriteLn('Nici un nume de fisier in linia de comanda.');
         Write('Dati numele primului fisier    : '); ReadLn(nume_unu);
         Write('Dati numele celui de al doilea : '); ReadLn(nume_doi)
       end;
  Assign(F,nume_unu); Append(F); Assign(G,nume_doi); Reset(G);
  while not eof(G) do begin ReadLn(G,linie); WriteLn(F,linie) end;
  Close(G); Close(F); WriteLn('Am pus totul in ',nume_unu,'.'); ReadLn
end.
```

Dacă parametrii din linia de comadă ne interesează să-i vedem ca numere, va trebui să-i convertim din şiruri în valori numerice folosind procedura `Val` (prezentată la tipul `String`).

Procedurile `Erase` şi `Rename`. Sortarea unui fişier text

Să scriem un program care sortează un fişier text în el însuşi. Vom proceda în felul următor. Presupunând că fişierul text este destul de mare, vom prelua într-o listă de şiruri de caractere toate liniile sale (o variantă cu vectori a acestui program nu ar fi prea eficientă). Vom sorta lista (prin "*bubble-sort*"), apoi vom scrie şirurile într-un alt fişier. În final, vom şterge fişierul inițial şi-l vom redenumi pe cel de al doilea cu numele primului.

Un fişier `F` (neapărat închis) se şterge prin `Erase(F)`, iar redenumirea sa la `nume_nou` se face prin `Rename(F,nume_nou)`.

```
program SortareFisierText; { SORTARE.PAS }
type PCelula = ^Celula;
     Celula = record info: String; prec, urm: PCelula end;
     Lista = record inceput, curent,  sfirsit: PCelula; lung: Integer end;
procedure Init(var L: Lista);
begin
  with L do
       begin
         lung:=0; inceput:=nil; sfirsit:=nil; inceput^.urm:=sfirsit;
         inceput^.prec:=nil; sfirsit^.prec:=inceput; sfirsit^.urm:=nil;
         curent:=inceput
       end
end;
procedure PozitioneazaLaInceput(var L: Lista);
begin
  while not (L.curent=L.inceput^.urm) do L.curent:=L.curent^.prec
end;
procedure Adauga(var L: Lista; elem: String);
var C: PCelula;
begin
  New(C);
  with C^ do
       begin { sau L.sfirsit}
         info:=elem;prec:=L.sfirsit^.prec; urm:=L.sfirsit^.prec^.urm
       end;
  with L do
       begin
         sfirsit^.prec^.urm:=C; sfirsit^.prec:=C; lung:=lung+1;curent:=C
       end
end;
function Citeste(L: Lista): Boolean;
var F: Text; linie: String;
begin
  if ParamCount<1 then
     begin
       WriteLn('Utilizare SORTARE ', nume_fisier_text'); Citeste:=False
     end
  else
     begin
       Assign(F,ParamStr(1)); Reset(F); Init(L);
       while not eof(F) do begin ReadLn(F,linie); Adauga(L,linie) end;
       Close(F); Citeste:=True
     end
end;
procedure Scrie(L: Lista);
var F: Text;
begin
  PozitioneazaLaInceput(L); Assign(F,'TEMP.###'); ReWrite(F);
  while not (L.curent = L.sfirsit) do
        begin WriteLn(F,L.curent^.info); L.curent:=L.curent^.urm end;
```

```
  Close(F);
end;
procedure Sorteaza(var L: Lista);
var gata: Boolean; aux: String;
begin
  repeat
    gata:=True; { ma pozitionez la inceput si parcurg lista }
    PozitioneazaLaInceput(L);
    while not (L.curent^.urm = L.sfirsit) do begin
            if L.curent^.info < L.curent^.urm^.info then
               begin
                 gata:=False; aux:=L.curent^.info;
                 L.curent^.info := L.curent^.urm^.info;
                 L.curent^.urm^.info := aux
               end;
            L.curent:=L.curent^.urm
   end
  until gata
end;
procedure Redenumeste;
var F: Text;
begin
   Assign(F,ParamStr(1));Erase(F);Assign(F,'TEMP.###');Rename(F,ParamStr(1))
end;
var L: Lista;
begin
  if Citeste(L) then begin Sorteaza(L); Scrie(L); Redenumeste end
end.
```

4. Fişiere cu tip

Declarare

Pentru aplicații cu caracter economic, aceste fişiere sunt cele mai indicate a se folosi. Astfel, un **fişier cu tip** înseamnă o secvență de articole de acelaşi gen (de pildă nişte structuri complexe, nişte înregistrări), stocate pe disc.
Un fişier de un anumit tip `T` se declară prin:

```
var F: File of T
```

sau se defineşte un nou tip de date, declarându-se apoi o variabilă de acest tip:

```
type Fisier = File of  T;
var F: Fisier;
```

`T` nu poate fi un tip fişier sau un tip referință.

Exemple:

```
type elev = record
               nume: String;
               punctaj: Integer
            end;
     FisElevi = File of elev;
var F: FisElevi; G: File of String; H: File of Integer;
```

În declarațiile de mai înainte, `F` este un fişier ale cărui articole sunt de tip `elev`. De asemenea, `G` este un fişier de şiruri de caractere, însă nu e totuna cu un fişier de tip `Text`, iar `H` este un fişier de numere întregi.
Articolele pot fi accesate în mod direct, poziționarea pe articolului al `i`-lea făcându-se cu `Seek(F,i)`. De remarcat că articolele sunt numerotate începând cu `0`, iar numărul total de

articole ale fișierului este obținut prin funcția `FileSize(F)`. De asemenea, poziția curentă a pointerului de fișier (deci poziția curentă în fișier) se determină cu apelul `FilePos(F)`.

Operații

Un fișier cu tip `F` se asignează, ca orice fișier, prin `Assign(F,FISIER)`. Deschiderea pentru rescriere (sau crearea sa) se face cu `ReWrite(F)`, iar deschiderea pentru citire, dar și pentru scriere se face cu `Reset(F)`.

Scrierea efectivă de articole se face cu procedura `Write` și pot fi scrise mai multe articole simultan.

$$\texttt{Write(F, art}_1\texttt{, ..., art}_n\texttt{);}$$

După o scriere, în fișierul fizic nu vor apare informații inteligibile la o citire cu un editor de texte obișnuit. Singura posibilitate de a citi datele din fișier este cu o secvență de program corespunzătoare, ce recunoaște articolele scrise în fișier. Citirea se face cu procedura `Read`:

$$\texttt{Read(F, art}_1\texttt{, ..., art}_n\texttt{);.}$$

Procedura `AssignCRT`

În cele ce urmează vom reveni asupra programului cu admiterea în liceu a unor elevi. Vom combina posibilitățile listelor dinamice cu cele ale fișierelor. Propunem cititorului să studieze cu atenție acest program care poate constitui un model (în miniatură) pentru orice aplicație de acest gen, pe care el ar dori să o realizeze.

Din motive de economie, am considerat un elev având doar două câmpuri: *nume* și *punctaj*.

Programul are un meniu din care se pot selecta:

- *inițializarea bazei de date*: - se rescrie fișierul respectiv, se inițializează lista de elevi;
- *adăugarea unui elev*: - se citesc de la tastatură datele referitoare la un elev și elevul se adaugă listei de elevi
- *ștergerea unui elev*: - se caută (după nume) un elev în lista elevilor din memorie, iar dacă există, acesta este șters din listă
- *salvarea datelor într-un fișier*: - se scrie conținutul listei într-un fișier de date (de tip `file of elev`)
- *încărcarea datelor dintr-un fișier*: - se memorează conținutul unui fișier cu date despre elevi într-o listă
- *ordonarea după medii și listarea elevilor într-un fișier text*, ce ulterior poate fi imprimat cu o comandă `type nume_fisier > prn` pe hârtie, sau afișarea direct pe ecran a listei ordonate a elevilor;

Referitor la ultima operație din lista de mai înainte, trebuie să precizăm că am folosit în program un fișier `F` de tip text, care poate fi fie un fișier text fizic, real, fie chiar ecranul însuși. Pentru a lucra cu ecranul exact ca și cu un fișier text obișnuit, se face un apel **`AssignCrt(F)`**. Astfel, orice scriere în `F` va fi, de fapt, o afișare pe ecran. (La fel se poate face o citire de la tastatură.) Procedurile `ReWrite(F)` și `Close(F)` nu vor avea nici un efect asupra ecranului. Următorul program citește de la tastatură un șir, apoi îl afișează din nou pe ecran.

```
program Ecranul_FisierText;
uses Crt;
var F: Text; a: String;
begin
  AssignCrt(F); ReSet(F); ReadLn(F,a); Close(F);
  ReWrite(F); WriteLn(F,a); Close(F); ReadLn
end.
```

Controlul operațiilor de intrare și ieșire.

În unele aplicații cu fişiere, ca şi în cea de mai jos, nu se ştie dacă există un fişier sau nu, sau, dacă există, dacă o operație de citire din el (sau chiar o scriere în el) este realizată cu succes. Astfel, Turbo-Pascal ne pune la dispoziție o variabilă întreagă ce semnalează un eventual cod de eroare la operații de intrare-ieşire. Numele ei este **`IOResult`**, ea având valoarea zero când nu se semnalează nici o eroare de acest gen.
Putem controla, cu ajutorul acestei variabile, cazurile de eroare apărute într-o anumită secțiune dintr-un program. Totul este să folosim opțiunea de compilare **`{$I-}`** la începutul acelei secțiuni. Apoi verificăm dacă `IOResult` este zero sau nu, în caz negativ având de a face cu o eroare.

Un exemplu clasic de utilizare a celor spuse mai înainte este următoarea funcție booleană ce verifică existența unui fişier fizic pe disc, cu numele `nume`:

```
function ExistaFisier(nume: String): Boolean;
var F: File;
begin
     Assign(F,nume); {$I-}Reset(F){$I+};
     if IOResult=0 then begin Close(F); ExistaFisier := True end
     else ExistaFisier := False
end;
```

Ca exercițiu, lăsăm cititorului să aducă programului alte facilități, pe care noi, din economie de spațiu, nu le mai prezentăm aici (ordonare afabetică, căutare elev după o anumită cheie). Propunem, de asemenea, realizarea unei interfețe prietenoase, ce să permită alegerea unei operații din meniu fie cu tastatura, fie cu mouse-ul, fie simultan.

```
{$X+}
program AdmitereInLiceu_fisiere_si_liste;
uses Crt;
type elev = record nume: String; punctaj: Integer end;
     FisElevi = file of elev; Localizare = (tastatura, ecran, fisier);
     tip_info=elev;
     PCelula = ^Celula;
     Celula = record info: tip_info; prec, urm: PCelula end;
     Lista = record inceput, curent,  sfirsit: PCelula; lung: Integer end;
var BazaDeDate: FisElevi;
procedure Init(var L: Lista);
begin
  with L do
       begin
         lung:=0; inceput:=nil; sfirsit:=nil; inceput^.urm:=sfirsit;
         inceput^.prec:=nil; sfirsit^.prec:=inceput; sfirsit^.urm:=nil;
         curent:=inceput
       end
end;
procedure PozitioneazaLaInceput(var L: Lista);
begin
  while not (L.curent=L.inceput^.urm) do L.curent:=L.curent^.prec
end;
procedure Adauga(var L: Lista; elem: tip_info);
var C: PCelula;
begin
  New(C);
  with C^ do
    begin info:=elem;prec:=L.sfirsit^.prec; urm:=L.sfirsit^.prec^.urm end;
  with L do
    begin lung:=lung+1;sfirsit^.prec^.urm:=C;sfirsit^.prec:=C;curent:=C end
end;
procedure Sterge(var L: Lista);
var C: PCelula;
begin
```

```
  with L do
       begin
         C:=curent; C^.prec^.urm:=C^.urm; C^.urm^.prec:=C^.prec;
         curent:=C^.urm; dispose(C); lung:=lung-1
       end
end;
procedure Curata(var L: Lista);
var E: tip_info; C: PCelula;
begin
  PozitioneazaLaInceput(L);
  while L.lung>0 do
    with L do
      begin
        C:=curent; C^.prec^.urm:=C^.urm; C^.urm^.prec:=C^.prec;
        curent:=C^.urm; Dispose(C); lung:=lung-1
      end;
  Init(L)
end;
procedure Afis(var L: Lista; unde: Localizare; Fis: String);
var i: Integer; F: Text;
begin
  if unde = fisier then Assign(F,Fis)
  else AssignCrt(F);
  ReWrite(F); PozitioneazaLaInceput(L);
  i:=1;
  while not (L.curent = L.sfirsit) do
        begin
          with L.curent^.info do WriteLn(F,i,'. ',nume:20,punctaj:5);
          L.curent:=L.curent^.urm; i:=i+1
        end;
  WriteLn(F,'Total : ',i-1,' elevi.'); Close(F)
end;
procedure AdaugareElev(de_unde: Localizare; var L: Lista);
var E: elev;
begin
  if de_unde = tastatura then
     begin
       WriteLn('Adaugare elev');
       with E do
            begin
              Write('Dati numele    : '); ReadLn(nume);
              Write('Dati punctajul : '); ReadLn(punctaj)
            end
    end
  else
    begin
      {$I-}Read(BazaDeDate,E){$I+};
      if IOResult <> 0 then WriteLn('Eroare la citirea fisierului !')
    end;
  Adauga(L,E)
end;
procedure EliminareElev(var L: Lista);
var numele: String; gasit: Boolean;
begin
  WriteLn('Eliminare elev'); PozitioneazaLaInceput(L);
  WriteLn('Dati numele elevului : '); ReadLn(numele);
  gasit:=False;
  while (not (L.curent = L.sfirsit)) and (not gasit) do
    if L.curent^.info.nume=numele then gasit:=True
    else L.curent:=L.curent^.urm;
  if gasit then Sterge(L)
  else WriteLn('Nu exista acest elev...')
```

```
end;
procedure ListareEleviDupaPunctaj(var L: Lista);
var gata: Boolean; aux: elev; raspuns: String;
begin
  WriteLn('Listare elevi dupa punctaj');
  {ordonare elevi dupa medii: bubble-sort}
  repeat
    gata:=True; {pozitionez la inceput si parcurg lista}
    PozitioneazaLaInceput(L);
    while not (L.curent^.urm=L.sfirsit) do
      begin
        if L.curent^.info.punctaj < L.curent^.urm^.info.punctaj then
           begin
             gata := False; aux := L.curent^.info;
             L.curent^.info := L.curent^.urm^.info;
             L.curent^.urm^.info := aux
           end;
        L.curent:=L.curent^.urm
      end
  until gata;
  { afisare lista }
  WriteLn('Unde afisam ? ‘, ’[e = ecran / f = fisier]');
  if UpCase(ReadKey) = 'F' then
     begin
       Write('Dati numele fis. text : '); ReadLn(raspuns);
       Afis(L,fisier,raspuns+'.TXT')
     end
  else Afis(L,ecran,'')
end;
function ExistaFisier(nume: String): Boolean;
var F: File;
begin
  Assign(F,nume); {$I-}Reset(F){$I+};
  if IOResult=0 then begin Close(F); ExistaFisier := True end
  else ExistaFisier := False
end;
procedure IncarcareListaElevi(var L: Lista);
var E: elev; fis_date: String;
begin
  WriteLn('Incarcare date elevi'); Write('Dati fisierul (.DAT) -> ');
  ReadLn(fis_date);
  if not ExistaFisier(fis_date+'.DAT') then WriteLn('Fisier inexistent !')
  else
   begin
    Assign(BazaDeDate, fis_date+'.DAT'); Reset(BazaDeDate);
    {$I-}
    while (not eof(BazaDeDate)) and (IOResult=0) do AdaugareElev(fisier,L);
    if IOResult <> 0 then
       begin
         WriteLn('Eroare la ‘,‘citirea fisierului !');
         Write('Stergem lista deja’,‘creata ? [d/n]');
         if UpCase(ReadKey)='D' then Init(L)
       end;
    {$I+}
    Close(BazaDeDate)
   end
end;
procedure SalvareListaElevi(var L: Lista);
var fis_date: String;
  procedure Salvare;
  begin
    Assign(BazaDeDate, fis_date+'.DAT'); ReWrite(BazaDeDate);
```

```
    PozitioneazaLaInceput(L);
    while not (L.curent = L.sfirsit) do
       begin Write(BazaDeDate, L.curent^.info); L.curent:=L.curent^.urm end;
    WriteLn('Am salvat lista in fis. ', fis_date+'.DAT.'); Close(BazaDeDate)
  end;
begin
  WriteLn('Salvare date elevi'); Write('Dati fisierul (.DAT) -> ');
  ReadLn(fis_date);
  if ExistaFisier(fis_date+'.DAT') then
     begin
       WriteLn('Fisier existent ! ‘, ’Doriti rescriere ? [d/n] ');
       if UpCase(ReadKey)='D' then Salvare
     end
  else Salvare
end;
var L: Lista; optiune: Char;
begin
  TextColor(yellow); TextBackGround(blue); ClrScr; Init(L);
  repeat
    ClrScr; WriteLn; WriteLn; Write('ADMITEREA IN LICEU');
    WriteLn; WriteLn;
    WriteLn('           Meniu:'); WriteLn('           ******');
    WriteLn; WriteLn('1=initializare lista elevi');
    WriteLn('2=adaugare elev in lista, de la tastatura');
    WriteLn('3=stergere elev din lista');
    WriteLn('4=listare dupa punctaj pe ecran sau in fisier text');
    WriteLn('5=incarcare lista elevi din fisier de date');
    WriteLn('6=salvare lista elevi in fisier de date'); WriteLn;
    WriteLn('0=oprire program'); WriteLn;
    Write('Optiunea dvs. = '); optiune:=ReadKey; ClrScr;
    case optiune of
         '1': begin Curata(L); WriteLn('Lista initializata...') end;
         '2': AdaugareElev(tastatura,L); '3': EliminareElev(L);
         '4': ListareEleviDupaPunctaj(L); '5': IncarcareListaElevi(L);
         '6': SalvareListaElevi(L)
    end;
    if optiune in ['1'..'5'] then
       begin WriteLn; WriteLn('Apasati o tasta !'); ReadKey end
  until optiune='0'
end.
```

Observație. Trebuie precizat că acest program are un caracter pur didactic, el neputând fi folosit într-o situație reală (un examen real de admitere în liceu) decât pentru ordonarea descrescătoare, în ordinea mediilor, a elevilor. Deşi acest lucru este foarte util, pentru gruparea pe clase, în funcție de opțiunile elevilor, de numărul de locuri disponibile la fiecare profil, precum şi în funcție de limita inferioară (5), programul va trebui îmbunătățit, mai ales că ar fi bine ca să fie gestionate şi notele elevilor, nu doar mediile lor (care aici sunt numere întregi).

Sortarea unui fişier cu tip

Fie un fişier de date creat cu opțiunea `6` (salvare date elevi) din programul anterior. Să ordonăm acest fişier, fără a memora într-o listă conținutul fişierului. Pentru aceasta, vom folosi poziționarea în fişier cu ajutorul procedurii `Seek`. Vom privi, astfel, fişierul (`F`) ca pe un vector de lungime `FileSize(F)`. Metoda utilizată este “*quick-sort*”. O asemenea modalitate de ordonare a conținutului unui fişier se dovedeşte foarte util în cazul fişierelor foarte mari!

```
program SortareFisierDAT;
type elev = record nume: String; punctaj: Integer end;
     FisElevi = file of elev;
var F: FisElevi; E: elev; n: LongInt; fisier: String;
```

```
procedure Pozitioneaza(start, finis: LongInt; var k: LongInt);
var i,j: LongInt; d: ShortInt; E_i,E_j: elev;
begin
  d:=0; i:=start; j:= finis;
  while i<j do
    begin
      Seek(F,i); Read(F,E_i); Seek(F,j); Read(F,E_j);
      if E_i.punctaj < E_j.punctaj then
         begin
           Seek(F,j); Write(F,E_i); Seek(F,i); Write(F,E_j); d:=1-d
         end;
      Inc(i,d); Dec(j,1-d)
    end;
  k:=i
end;
procedure Quick(inceput, sfarsit: LongInt);
var k: LongInt;
begin
  if inceput<sfarsit then
    begin
      Pozitioneaza(inceput,sfarsit,k);Quick(inceput,k-1);Quick(k+1,sfarsit)
    end
end;
begin
  if ParamCount<1 then
     begin
       Write('Dati nume fis. (.DAT) > '); ReadLn(fisier)
     end
  else fisier:=ParamStr(1);
  { se accepta si fisiere cu tot cu extensie si fara extensie }
  if Pos('.',fisier)=0 then   fisier:=fisier+'.DAT';
  Assign(F,fisier); Reset(F); WriteLn('Continut initial fis.',fisier);
  n:=0;
  while not eof(F) do
    begin
      n:=n+1; Read(F,E); WriteLn(n:5,'. ',E.nume:20,' ',E.punctaj:5)
    end;
  WriteLn('Total : ',n,' articole.');
  Quick(0,FileSize(F)-1);
  WriteLn; WriteLn('Continut actual fis. ',fisier);
  n:=0; Seek(F,0);
  while not eof(F) do
    begin
      n:=n+1; Read(F,E); WriteLn(n:5,'. ',E.nume:20,' ',E.punctaj:5)
    end;
  WriteLn('Total : ',n,' articole.'); Close(F); ReadLn
end.
```

Un exemplu de execuție a acestui program este dat mai jos. (Baza de date a fost creată cu programul din paragraful anterior !)

```
Dati numele fisierului (.DAT) -> alfa
Continut initial fisier alfa.DAT :
    1.      Ionescu Daniel     7
    2.      Vasilescu Marin    8
    3.      Popa Claudia       5
    4.      Anton Emilia       6
    5.      Done Cristian      8
Total : 5 articole.
Continut actual fisier alfa.DAT :
    1.      Done Cristian      8
    2.      Vasilescu Marin    8
```

```
    3.      Ionescu Daniel     7
    4.      Anton Emilia       6
    5.      Popa Claudia       5
Total : 5 articole.
```

Am pus în evidență, în textul programului anterior, două lucruri:

* o tehică de a putea porni programul cu numele fişierului atât cu extensie, cât şi fără extensie, extensia considerându-se implicită `DAT`, aici, lucru ce va putea fi folosit cu succes în cazuri asemănătoare;
* apelul procedurii `Quick` de sortare, în care se observă că se face o ordonare începând cu prima poziție (cu numărul de ordine `0`) şi terminând cu ultima poziție a fişierului (cu numărul de ordine dat de lungimea acestuia, mai puțin o unitate); orice alte numere aici, care însă trebuie să fie de tip întreg lung (`LongInt`) ar ordona fişierul de date între respectivele poziții.

5. Fişiere fără tip

Despre fişierele fără tip, în afară de faptul că se declară prin

```
var F: File;
```

tot ce am prezentat în primul paragraf rămâne valabil, în continuare. Fireşte, procedurile `Assign`, `Reset` (pentru deschidere în citire-scriere), `ReWrite` (pentru creare, inițializare, rescriere) şi `Erase` (pentru ştergere) sau `Rename` (redenumire) sunt şi ele aplicabile. `FilePos(F)` ne dă poziția curentă (byte-ul curent) din fişier, iar cu `Seek(F,poz)` ajungem pe o anumită poziție, adică pe un anumit byte din fişierul `F`. De această dată şi `FileSize(F)` ne va da lungimea fişierului în *bytes*, deci mărimea sa absolută în octeți. Ele sunt cea mai generală formă de stocare a datelor pe suport magnetic de mare capacitate, iar citirea şi scrierea se face cu procedurile **`BlockRead`** şi, respectiv, **`BlockWrite`**, care pot scrie blocuri de date întregi.

- **Citirea** unui număr de blocuri de date dintr-un fişier fără tip în memorie:

```
procedure BlockRead(var F:File; var Buf;Count:Word [;var Result:Word]);
```

unde:

`F` = fişierul fără tip;

`Buf` = o variabilă oarecare, care va fi tamponul (*buffer*-ul) fişierului (procedura se va folosi de adresa acestei variabile). Blocurile citite vor fi depuse în memorie unul după altul, începând cu această adresă.

`Count` = numărul maxim de blocuri ce urmează a fi citite din fişierul `F`;

`Result`: Dacă în apel nu se specifică parametrul efectiv corespunzător lui `Result`, iar numărul de blocuri citite este mai mic decât `Count`, se provoacă o eroare de intrare-ieşire. Dacă, însă, `Result` are drept corespondent o variabilă de tip `Word`, atunci în această variabilă se va găsi numărul de blocuri citite efectiv. Dimensiunea unui bloc este implicit de 128 octeți, dar ea se poate modifica (până la valoarea 65535), printr-un apel special al procedurii `Reset` sub forma: `Reset(F,marime_bloc)`.

- **Scrierea** unui număr de blocuri de date din memorie într-un fişier fără tip se face cu procedura:

```
procedure BlockWrite(var F:File;var Buf;Count:Word [;var Result:Word]);
```

unde:
`F` = fişierul fără tip;
`Buf` = o variabilă oarecare (procedura va utiliza adresa acestei variabile). Blocurile consecutive din memorie începând de la adresa variabilei `Buf` vor fi scrise în fişierul `F`;
`Count` = numărul maxim de blocuri ce urmează a fi scrise în fişierul `F`;
`Result`: Dacă nu există în apel, iar numărul de blocuri scrise este mai mic decât `Count`, atunci se produce o eroare de intrare-ieşire. Dacă există, în variabila trimisă ca parametru efectib se va afla, la sfârşit, numărul de blocuri scrise efectiv.
Şi procedura `ReWrite` poate fi apelată cu numărul de mărime al unui bloc (articol, înregistrare): `ReWrite(f,marime_bloc)`. Aceste două proceduri se pot aplica doar fişierelor fără tip, care sunt, desigur, deschise.

Folosirea practică a acestor proceduri va fi prezentată în cadrul exemplelor de la capitolul de grafică, acolo unde vom vorbi despre diferitele formate ale fişierelor grafice, adică a acelor fişiere conținând imagini, filme etc., fişiere cu informații de cele mai diferite tipuri.
Deocamdată vom prezenta un program care copiază un fişier în altul, citind câte un caracter din primul şi scriindu-l în cel de al doilea. Fişierele sursă şi destinație vor trebui specificate în linia de comandă (sau în meniul `Run`, opțiunea `Parameters...`, după cum am mai spus.).

```
program Copie;
var F, G: file; OctetiCititi, OctetiScrisi: Word;
    Buf: array[1..2048] of Char; sursa, dest: String;
begin
  if ParamCount<2 then WriteLn('Utilizare: COPIE fis_sursa fis_dest')
  else
    begin
      sursa:=ParamStr(1); dest:=ParamStr(2); { marimea unui bloc = 1 }
      Assign(F,sursa); Reset(F,1); Assign(G,dest); ReWrite(G,1);
      Writeln('Copiem ', FileSize(F), ' octeti...');
      repeat
        BlockRead(F, Buf, SizeOf(Buf), OctetiCititi);
        BlockWrite(G, Buf, OctetiCititi, OctetiScrisi);
      until (OctetiCititi = 0) or (OctetiScrisi <> OctetiCititi);
      Close(F); Close(G)
    end
end.
```

În textul programului anterior apare o funcție nouă, **`SizeOf`**, care returnează mărimea, în octeți, ocupată în memorie de o anumită variabilă, indiferent de tipul ei. În cazul nostru, pentru vectorul `buf` avem `SizeOf(Buf) = 2048`.

6. Exerciții recapitulative

1. Să se scrie un program care tipăreşte distribuția frecvenței lungimii cuvintelor aflate într-un fişier text. Cuvintele sunt separate prin spații.
2. Să se scrie un program care compară, linie cu linie, conținutul a două fişiere text, afişând numărul de ordine al celor neidentice.
3. Se dau două şiruri de caractere `S` şi `T` şi un fişier text `F`. Să se scrie procedura care copiază tot conținutul lui `F` într-un alt fişier text `G`, însă înlocuieşte peste tot şirul `S` cu şirul `T`.
4. Să se scrie un program care să permită administrarea unui top al preferințelor dumneavoastă muzicale. Se vor folosi fişiere cu tip.
5. Să se concateneze două fişiere oarecare într-un al treilea fişier. Numele celor trei fişiere se vor citi din linia de comandă a programului.
6. Să se realizeze un program pentru o agendă telefonică care să permită gestionarea unor persoane, a adreselor lor şi a numerelor lor de telefon. Operațiile care vor fi realizate sunt: adăugare persoană, eliminare persoană, căutare numar de telefon a unei persoane, căutarea unei persoane a cărui număr de telefon se cunoaşte, modificare număr de telefon persoană, ordonare alfabetică, listarea persoanelor într-un fişier text.

Lecția 10
BIBLIOTECI

1. Generalități. Bibliotecile standard

Fie că dorim să scriem programe mari (mai mari de 64 Ko), fie că vrem să le modularizăm şi mai mult, avem nevoie să folosim **biblioteci** sau **unități de program** sau **unit**-uri. Un **unit** este o colecție de constante, declarații de tip şi de variabile, declarații şi definiții de proceduri şi funcții, scrisă într-un alt fişier sursă diferit de cel al programului, care poate fi compilată separat (obținându-se un fişier `TPU`) şi care poate fi utilizată de un program principal, sau un alt unit, prin specificarea numelui său într-o clauză **uses**.

În afară de avantajele "tehnice" pe care le aduce ideea de unit, există şi avantaje de programare. De pildă, dacă am scris anumite funcții şi proceduri, pe care le folosim în mod frecvent în legătură cu un anumit gen de lucrări, în programe, e bine ca să le punem pe toate într-o bibliotecă, pentru a nu le rescrie odată cu fiecare program ce le utilizează.

Există două feluri de biblioteci. Cele gata compilate, standard, livrate odată cu mediul Turbo-Pascal, şi cele pe care le putem creea noi. Cele standard se găsesc fie în subdirectorul `UNITS` al directorului în care se află limbajul Turbo-Pascal (de exemplu `GRAPH.TPU`), fie se găsesc, în fişierul colecție de biblioteci **`TURBO.TPL`** (ca, de pildă, unit-urile `System`, `Dos` şi `Overlay`). **`System`** este un unit special care se încorporează în orice program Pascal. El cuprinde toate elementele de bază ale limbajului.

Despre unele unit-uri am vorbit deja, parțial:

- **`Crt`** permite utilizarea funcțiilor şi procedurilor referitoare la lucrul cu ecranul în modul text, precum şi cu tastatura sau difuzorul;
- **`Dos`** conține proceduri şi funcții echivalente cu apelurile sistemului de operare `DOS` cele mai uzuale (citire de dată şi timp, dimensiune disc, diferite funcții referitoare la fişiere.

Există, de asemenea, unit-ul **`Graph`**, care conține subprograme grafice. Acestea au nevoie, însă, şi de informații din fişierele `BGI` şi `CHR`, situate în subdirectorul `BGI` al mediului de programare. Despre biblioteca subprogramelor grafice vom vorbi în capitolul următor.
Unit-ul **`Printer`** permite folosirea imprimantei ca pe un fişier text, cu numele **LST**, ce nu mai necesită deschidere sau închidere. Mai jos este dat un exemplu de scriere la imprimantă (presupunând că ea este pregătită, are hârtie etc.).

```
program ScriereLaImprimanta;
uses Printer;
begin
  WriteLn(Lst, 'Scriere la imprimantă !!!')
end.
```

În paragraful al treilea va fi dată o aplicație a unit-ului Printer.
Există şi două unit-uri ce permit folosirea programelor grafice din versiunea 3.0 a limbajului. Acestea sunt `Turbo3` şi `Graph3`.

2. Biblioteci create de programator

Forma generală

Un programator îşi poate crea propriile biblioteci, care să se folosească întocmai celor standard, predefinite. Aceste unit-uri vor fi scrise în fişiere diferite. Există însă o restricție (ce nu apare la programe): numele fişierului să fie identic cu identificatorul de unit!

Forma generală a unui unit este asemănătoare celei a unui program, însă există unele particularități.

```
unit nume; { obligatoriu în fisierul nume.pas !! }
    interface
     elemente_recunoscute_in_exterior;
    implementation
     elemente_interne_bibliotecii;
begin
    sectiunea_de_initializare
end.
```

Componentele unui unit

Aşadar, un unit se compune din trei părți, prima şi a treia fiind opționale. Dacă a treia lipseşte, poate lipsi şi cuvântul `begin`.

- Secțiunea de interfață începe cu cuvântul `interface`. Ea cuprinde declarații de subprograme ce sunt necesare a fi recunoscute de programele sau bibliotecile exterioare, care utilizează acest unit. De asemenea, aici pot fi declarate şi definite constante, variabile sau tipuri de date comune unit-ului şi programului (sau unit-urilor) care îl foloseşte.
Dacă în aceste declarații se utilizează elemente dintr-un alt unit, atunci se va specifica acest lucru printr-o cauză `uses`.

- Secțiunea de implementare începe cu cuvântul `implementation`. Ea va cuprinde definirea explicită a subprogramelor declarate în secțiunea de interfață. De asemenea, pot apare aici şi alte subprograme sau variabile, constante, tipuri, locale unit-ului, necesare acestuia pentru a-i realiza procedurile sau funcțiile scrise în secțiunea de interfață.
Çi aici poate apare clauza `uses`, indicând faptul că se utilizează elemente din alte biblioteci.
În secțiunea de implementare, nu e obligatoriu ca funcțiile declarate în interfață să aibă scris din nou listele de parametri; nu contează nici ordinea lor.

- Secțiunea de inițializare începe, dacă există, cu cuvântul `begin`. Ea conține nişte instrucțiuni ce permit efectuarea inițializării variabilelor definite în secțiunea de interfață. Pot apare, însă, şi alt gen de instrucțiuni. Când se execută un program care utilizează respectivul unit, înainte de a intra în corpul programului principal, se execută aceste instrucțiuni din unit. Dacă programul foloseşte mai multe unit-uri, execută porțiunile de inițializare a tuturor unit-urilor, în ordinea corespunzătoare celei din clauza `uses`.

O bibliotecă pentru lucrul cu vectori

Vom crea, în fişierul `UVEC.PAS` un unit, cu numele `uVec`, care să conțină trei proceduri care să execute citirea, afişarea şi ordonarea unui vector de numere întregi. Aceste trei proceduri vor apare, alături de tipul `vector` şi constanta `NMax` în partea de interfață a unit-ului. Ele vor fi definite în partea de implementare. Tot aici va apare şi o procedură locală unit-ului, deci ce nu va fi recunoscută în exterior, în programul principal. Această procedură realizează interschimbarea valorilor a două variabile întregi.

```
unit uVec; { neapărat în fisierul UVEC.PAS !! }
{ operatii simple cu vectori de numere intregi }

interface
      const NMax = 10; { lungimea maxima a unui vector }
      type vector = array[1..NMax] of Integer;
      procedure Citeste(var V: vector; var n: Integer);
      procedure Scrie(V: vector; n: Integer);
      procedure Ordoneaza(var V: vector; n: Integer);
implementation
```

```
var i,j: Integer; { variabile locale bibliotecii }
procedure Citeste;
begin
  repeat
    Write('Dati lungimea vectorului : ');
    ReadLn(n)
  until n in [1..NMax];
  for i:=1 to n do
      begin
        Write('Dati elem. nr. ',i,' : '); ReadLn(V[i])
      end
end;
procedure Scrie;
begin
  for i:=1 to n-1 do Write(V[i],', ');
  WriteLn(V[n])
end;
procedure Schimba(var x,y: Integer);{procedura locala in unit}
var aux: Integer; { locala lui Schimba, nu si in unit ! }
begin aux:=x; x:=y; y:=aux; end;
procedure Ordoneaza;
begin
  for i:=1 to n-1 do
   for j:=i+1 to n do
    if V[j]<V[i] then Schimba(V[i],V[j]){apel procedura locala}
end;
end.
```

Un exemplu de utilizare a celor definite în unit-ul anterior este programul următor:

```
program TestUnitVectori;
uses Crt, uVec; { se utilizeaza un unit standard si unul propriu }
var X: vector; { tipul "vector" este preluat din unit-ul "uVec" }
    NrElemente: Integer;
begin
  WriteLn('** Ordonare vector **');  WriteLn('  - Citirea elementelor:');
  Citeste(X,NrElemente); { apel procedura din unit-ul "uVec" }
  WriteLn('  - Elementele ordonate sunt:'); Ordoneaza(X,NrElemente);
  Scrie(X,NrElemente); ReadLn
end.
```

Propunem cititorului, ca exerciţiu, să îmbunătăţească această bibliotecă cu noi funcţii şi proceduri necesare lucrului cu vectori.

Observaţii

1. Dacă un program utilizează două unit-uri (`unit1` şi `unit2`) ce cuprind, fiecare, o funcţie `FUN`, pentru a preciza că ne referim la funcţia `FUN` din unit-ul `unit1` vom scrie `unit1.FUN`, ca la articolele unei înregistrări. Astfel, dacă într-un unit definim o funcţie `Close`, pentru a ne referi la proceura `Close` standard, în cazuri de ambiguitate, se va scrie `System.Close`.

2. Fie un program `PROG` şi două unit-uri `Unit1` şi `Unit2`, de forma:

```
program PROG;          unit Unit1;           unit Unit2;
uses Unit2;            interface             interface
...........            .............         uses Unit1;
begin                  implememtation        ..............
end.                   end.                  implementation
                                             ..............
                                             end.
```

În cazul de mai sus, `Unit2` utilizează `Unit1`, iar programul `PROG` utilizează pe `Unit1`. Dacă în programul principal apare doar `uses Unit2`, programul `PROG` poate să refere doar

identificatorii din secțiunea de interfață a lui `Unit2`, chiar dacă `Unit2` are acces la `Unit1`, iar în cazul în care în `PROG` avem clauza `uses Unit1, Unit2`, atunci el (`PROG`) poate să refere atât identificatorii din secțiunea de interfață a lui `Unit2`, cât şi pe cei din secțiunea de interfață a lui `Unit1`.

3. Aplicații

Listarea la imprimantă din program

O aplicație imediată a unit-ului standard `Printer` este folosirea imprimantei pentru a lista, dintr-un program, un fişier text oarecare. Ideea este simplă. Se deschide respectivul fişier text, al cărui nume ar putea fi preluat din linia de comandă. Se citeşte linie cu line conținutul acestuia şi se scrie în fişierul standard `LST`, care reprezintă imprimanta. La fiecare 60 de rânduri, se trimite codul #12 (`Form Feed`) imprimantei, adică se trece la o linie nouă. De asemenea, la începutul fiecărei pagini se imprimă un antet ce cuprinde numărul de pagină. Variabila `ModTiparire`, cu valoarea inițială `#18`, indică cum să se facă tipărirea. Astfel, pentru `#15`, se obține o listare cu litere mai mici.

Putem verifica dacă imprimanta este pregătită pentru listare trimițând un caracter neafişabil (`#7` = sunet) şi testând valoarea variabilei `IOResult`, care dacă nu este zero, indică o eroare de intrare-ieşire. Funcția este similară celei ce verifică existența unui fişier pe disc, pe care am explicat-o în lecția 9 şi se regăseşte în programul de mai jos.

```
program ListareLaImprimanta; { in fisierul IMPRIM.PAS }
uses Printer,Crt;
const ModTiparire: Char=#18; NrRanduriPePagina = 60;
var linie: String; fis: Text; NumeFis: String[79];
      rand: Byte; cod: Integer;  NrPagina: Word;
function ExistaFisier(NumeFis: String): Boolean;
var F: File;
begin
  Assign(F,NumeFis);
  {$I-} Reset(F) {$I+};
  if IOResult = 0 then begin Close(F); ExistaFisier := True end
  else ExistaFisier := False
end;
function ImprimantaEsteGata: Boolean;
begin
  {$I-} Write(Lst,#7); {$I+}
  ImprimantaEsteGata := IOResult=0
end;
function Antet: String;
var Sir: String;
begin
  Str(NrPagina:4,Sir); Antet:='          pagina '+Sir
end;
procedure Tipareste;
begin
  while not ImprimantaEsteGata do
    begin
      WriteLn('Imprimanta nu este pregatita!');
      WriteLn('Esc = abandon, Enter = start');
      if ReadKey = #27 then Exit
    end;
  Assign(fis,NumeFis); Reset(fis); WriteLn(Lst,Antet); WriteLn(Lst);
  WriteLn(Lst);
  repeat
    if rand = NrRanduriPePagina then
```

```
       begin
         rand := 0; Inc(NrPagina); Write(Lst,#12); WriteLn(Lst);
         WriteLn(Lst,Antet); WriteLn(Lst); WriteLn(Lst)
       end;
    Inc(rand); ReadLn(fis,linie); WriteLn(Lst,linie);
  until eof(fis);
  Close(fis); Write(Lst,#12)
end;
begin
  WriteLn; WriteLn; WriteLn('* Imprimare fisier text *'); WriteLn;
  if ParamCount<1 then
     WriteLn('Utilizare: IMPRIM nume_fis [pagina_start] .')
  else
     begin
       NumeFis := ParamStr(1);
       if ParamCount = 2 then Val(ParamStr(2),NrPagina,cod)
       else NrPagina := 1;
       rand := 1; WriteLn('Mod normal sau mic ','de tiparire [n/m] ? ');
       if UpCase(ReadKey) = 'M' then ModTiparire := #15;
       if ExistaFisier(NumeFis) and (ParamCount>0) then Tipareste
       else WriteLn('Nu exista acest fis. !')
     end
end.
```

Lucrul cu mouse-ul şi cursorul în modul text

În secțiunea 3 din lecția 6 am vorbit despre unele proceduri de lucru cu mouse-ul în modul text. Atunci, ansamblul lor era notat cu `M` şi era rescris în orice program ce le utiliza. De această dată le putem grupa într-un unit. Acest unit cuprinde, de asemenea, procedurile ce permit ascunderea sau afişarea cursorului pâlpâitor (vezi programul cu turnurile din Hanoi, din lecția 5). În continuarea unit-ului este dat un mic program demonstrativ ce desenează cerculeţe în modul text, cu mouse-ul. { `utmouse.pas` }

```
unit uTMouse; { unit de lucru cu mouse-ul si cursorul in modul text }
interface
      procedure HideCursor; { ascunde cursorul palpitor }
      procedure ShowCursor; { il arata }
      procedure MouseInit;  { initializare mouse }
      procedure MouseShow;  { afisare mouse }
      procedure MouseHide;  { ascundere mouse }
      procedure MouseTData(var buton, x, y: Integer); {preia date mouse}
implementation
        procedure HideCursor; assembler;
        asm mov ax,$0100; mov cx,$2607; int $10 end;
        procedure ShowCursor; assembler;
        asm mov ax,$0100; mov cx,$0506; int $10 end;
        procedure MouseInit; assembler;
        asm mov ax,0; int $33 end;
        procedure MouseShow; assembler;
        asm mov ax,1; int $33 end;
        procedure MouseHide; assembler;
        asm mov ax,2; int $33 end;
        procedure MouseTData;
        var bb,xx,yy: Integer;
        begin
          asm mov ax,3; int $33; mov bb,bx; mov xx,cx; mov yy,dx end;
          buton := bb; x := xx div 8 + 1; y := yy div 8 + 1
        end;
begin
  { Initiliazari }
```

```
  ShowCursor; MouseInit; MouseShow
end.
```

Următorul program afişează litera “o” prin apăsarea butonului din stânga al mouse-ului; la apasarea butonului din dreapta, programul se opreşte; pe parcursul desenării, cursorul pâlpâitor nu se vede, ci doar cel corespunzător mouse-ului.

```
program TestMouseSiCursor;
uses Crt,uTMouse;
{ se vor apela mai intai procedurile din sectiunea de initializare
  a lui uTMouse ! }
var b,x,y: Integer;
begin
  TextBackGround(0); TextColor(LightBlue);
  ClrScr; HideCursor;
  repeat
    MouseTData(b,x,y);
    if b=1 then
       begin
         MouseHide; GoToXY(x,y); Write('o');
         MouseShow
       end
  until b=2;
  TextColor(7); ShowCursor
end.
```

Majoritatea programelor mari pe care le veți realiza e bine să folosească unit-uri. Aşa vom proceda şi noi în capitolele ce urmează.
Ca exerciții, propunem cititorului să realizeze unit-uri pentru diferite operații cu şiruri de caractere, pentru proceduri de sortare, generare de mulțimi, algoritmi referitori la grafuri, etc.. Se vor folosi procedurile descrise în lecțiile 5,6,7 şi 8.

Lecția 11
GRAFICĂ ŞI ANIMAȚIE

1. Noțiuni introductive

Să învățăm singuri principalele proceduri grafice!
Toate exemplele din lecțiile anterioare foloseau modul text de afişare (cu excepția programului de derivare formală). Ecranul calculatorului poate fi folosit şi în **modul grafic**, reprezentat de o suprafață de puncte numite **pixeli**. Pe perioada folosirii modului grafic, nu se mai dispune, însă, de procedurile unit-ului `Crt` care se referă la afişări pe ecran (`ClrScr`, `GoToXY`, `TextColor`). De asemenea, nu se recomandă folosirea lui `ReadLn` decât fără parametri, sau a lui `Write`, `WriteLn`. Afişările se realizează cu alte proceduri în modul grafic. Rămân disponibile, însă, `ReadKey`, `KeyPressed` şi cele referitoare la sunet (`Sound`, `Delay`, `NoSound`). Funcțiile `ReadKey` şi `KeyPressed` vor fi folosite pentru a scrie proceduri de introducere de date în modul grafic.

În modul grafic se poate lucra, uneori, cu mai multe pagini video, adică se pot afişa alternativ, mai multe ecrane grafice. Ecranul are un număr de pixeli, numerotați pe orizontală şi verticală, începând cu colțul stânga-sus, în dreapta, respectiv în jos. Astfel, fiecare pixel are două coordonate, abscisa (orizontala) şi ordonata (verticala), colțul stânga-sus având coordonatele (`0,0`). Cel de dreapta jos are coordonatele în funcție de **placa grafică** folosită (un dispozitiv intern al calculatorului), precum şi de **modul grafic** (un anumit mod de organizare a ecranului, dat de placa grafică) şi de **driver-ul grafic** (un fişier cu extensia `BGI`) folosite.

Astfel, pentru calculatoarele cele mai utilizate astăzi - în instituții, şcoli sau întreprinderi - cu siguranță aveți o placă grafică ce acceptă 640 de pixeli pe orizontală, 480 pe verticală şi un număr de 16 culori. Aceste numere pot fi aflate, totuşi, cu funcțiile: `GetMaxX`, `GetMaxY`, respectiv `GetMaxColor`.

`GetMaxX` ne dă abscisa maximă, iar **`GetMaxY`** ne dă ordonata maximă. Astfel, la o rezoluție a ecranului de `640×480` pixeli, vom obține `639`, respectiv `479`, căci numerotările încep de la zero. **`GetMaxColor`** ne dă numărul ultimei culori, pentru 16 culori vom obține `15`, culorile fiind numerotate de la `0` la `15`.

De fapt, există 16 culori în acest mod, numit VGA. Lor le corespund nişte constante întregi predefinite: `Black`, `Blue`, `Red`, `Magenta`, `Green`, `Brown`, `Cyan`, `DarkGray`, `LightGray`, `LightBlue`, `LightCyan`, `LightMagenta`, `LightRed`, `LightGreen`, `Yellow`, `White`. De reținut că `Black=0`, iar `White=15`.

Trebuie precizat că pe ecran pot apare, în unele moduri grafice, mai multe pagini grafice; acest lucru va fi înțeles cu ajutorul unor exemple, mai târziu.
Pentru a realiza programe de grafică, trebuie să urmați următorii paşi:

- să vă asigurați că dispuneți de biblioteca `GRAPH.TPU`, care trebuie să se afle în subdirectorul `UNITS`;
- să verificați dacă dispuneți de fişierele cu extensiile `BGI` şi `CHR` în subdirectorul `BGI`; aceste fişiere reprezintă driverele grafice, respectiv nişte fişiere ce conțin, codificat, diferite tipuri (fonturi) de caractere afişabile în modul grafic;
- să declarați `uses Graph` în programul principal;

- să scrieți o procedură (sau o secvență de program) care să treacă ecranul în modul grafic, de forma:

```
procedure OpenGraphGenerala;
var DriverGrafic, ModGrafic: Integer;
begin
   DriverGrafic := valoare;
   [ ModGrafic := valoare_1; { eventual } ]
   InitGraph(DriverGrafic, ModGrafic, cale_către_driver_ul_grafic_BGI)
end;
```

- să scrieți programul propriu-zis, apelând la procedurile grafice;
- să închideți modul grafic, apelând procedura `CloseGraph`.

În procedura de mai sus, dacă se scrie constanta `Detect` (cu valoarea zero) în locul variabilei `valoare`, iar rândul suplimentar nu apare, atunci se va intra în cea mai mare rezoluție posibilă, în funcție de driver-ele grafice disponibile. Dacă nu se foloseşte valoarea 0 (constanta `Detect`), atunci se pot folosi alte variante, conform tabelului de mai jos:
Acest tabel este, în realitate, mult mai complex, însă acestea sunt cele mai utilizate moduri grafic. În prezenta lucrare noi vom considera implicit modul grafic `640×480×16` culori `× 1` pagină video, toate exemplele din carte referindu-se la el.

Driver grafic		Mod grafic		Rezoluție	Număr culori	Număr pagini
Constantă	Cod	Constantă	Cod			
EGA	3	EGALo	0	640×200	16	4
EGA	3	EGAHi	1	640×350	16	2
VGA	9	VGALo	0	640×200	16	4
VGA	9	VGAMed	1	640×350	16	2
VGA	9	VGAHi	2	640×480	16	1

Doar în paragraful în care se va prezenta un program special de transformări de imagini vom folosi modul grafic `640×350×16×2`, iar atunci procedura de inițializare grafică va fi alta, decât cea uzuală, pe care o dăm mai jos:

```
procedure OpenGraph;
var gd,gm: Integer;
begin
  gd := Detect;
  InitGraph(gd,gm,'C:\BP\BGI')
end;
```

După cum vedeți, am considerat implicit calea grafică '`C:\BP\BGI`', care este în calculatorul pe care s-au testat exemplele din carte. Dumneavoastră veți face modificările necesare, pentru a adapta programele la calculatoarele dumneavoastră.

Spuneam că dispunem de 16 culori. Acestea pot fi folosite atât pentru fundal, cât şi pentru desenare. Culoarea fundalului se alege cu **`SetBkColor(culoare)`**, iar cea pentru trasarea de linii, cercuri, dreptunghiuri sau elipse, prin **`SetColor(culoare)`**, unde `culoare` este un număr între `0` şi `15` sau o constantă corespunzătoare.

Doar punctele se desenează altfel: **`PutPixel(x,y,c)`** va aprinde un pixel de coordonate `(x,y)`, în culoarea `c`.

Rulați următorul program, el vă va lămuri în privința celor de mai înainte. (De fapt, chiar firma *Borland* vă pune la dispoziție un program, `BGIDEMO.PAS`, care reprezintă o demonstrație grafică excelentă a posibilităților drivere-lor `BGI` (*Borland Graphics Interface*) şi a unit-ului `Graph.TPU`.).

```
{$X+}
program PixeliSiFundal;
uses Crt,Graph;
procedure OpenGraph;
var gd,gm: Integer;
begin
  gd := Detect; InitGraph(gd,gm,'C:\BP\BGI')
end;
procedure Pixeli(x1,y1,x2,y2: Integer);
var i,j,c: Integer;
begin
  for i:=x1 to x2 do
    for j:=y1 to y2 do
        begin c:=Random(1+GetMaxColor); {16} PutPixel(i,j,c) end
end;
procedure DemoFundal;
var i: Byte;
begin
  for i:=1 to 10 do
      begin SetBkColor(Random(8)); Sound(i*10+100); Delay(50); NoSound end
end;
begin
     OpenGraph; Pixeli(100,70,450,300); DemoFundal;
     ReadKey; CloseGraph
end.
```

Pentru a realiza grafică vă trebuie unele cunoştințe de geometrie analitică, dar foarte puține! Cei ce au înțeles aceste noțiuni (coordonate, distanța între două puncte, ecuația cercului, a elipsei, translația, rotația etc.), vor vedea în acest capitol de grafică o adevărată colecție de jucării. Celorlalți le recomandăm să citească manualele de geometrie analitică (de pildă cel de clasa a XI-a), precum şi cărțile notate cu diez (#) la bibliografie. Ele îi vor lămuri asupra unor detalii de grafică.

Noi încercăm să învățăm cititorul prin exemple. Un pasionat de grafică va reuşi cu siuranță să înțeleagă aceste programe, să le modifice pentru a le face mai frumoase, mai atrăgătoare. Îi urăm succes!

Aşadar, să trecem în revistă unele subprograme grafice mai importante, restul le poate afla cititorul şi singur, dacă citeşte din `Help`-ul lui Turbo-Pascal. Pentru aceasta, va proceda astfel: va scrie `Graph`, apoi va apăsa `Ctrl-F1` şi va obține tot soiul de informații, pe care nu ar avea sens să le prezentăm aici. Totul e să cunoască un pic de engleză !
Să trecem, aşadar la lucru!

Pentru a stabili culoarea de desenare la valoarea `cul` se apelează, după cum am mai spus, **`SetColor(cul)`**.

Culoarea este valabilă - până la un nou `SetColor` - în desenarea dreptunghiurilor, a liniilor, elipselor şi cercurilor.

`SetWriteMode(modul)` ne alege modul de desenare, ce poate fi normal (`0`) sau conform operației logice "*sau exclusiv*" (`1`) (supradesenare înseamnă ştergere!), numit `XorPut`. Modul `XorPut` este valabil numai la linii şi la dreptunghiuri, nu şi la cercuri sau elipse.

Pentru a trasa o linie între punctele de coordonate `(x1,y1)` şi `(x2,y2)`, se va scrie **`Line(x1,y1,x2,y2)`**. Un dreptunghi având colțul stânga-sus `(x1,y1)` şi cel dreapta-jos `(x2,y2)` se va desena cu **`Rectangle(x1,y1,x2,y2)`**. Un cerc de centru `(x,y)` şi rază `r` se va realiza cu **`Circle(x,y,r)`**. O elipsă cu coordonatele centrului `(x,y)`, semiaxe `(a,b)` se va face cu **`Ellipse(x,y,0,360,a,b)`**. Vă imaginați că, de fapt, se desenează

un arc de elipsă, între unghiurile 0^o şi 360^o. Deci puteți modifica acele valori, să vedeți ce obțineți!

Puteți desena un dreptunghi umplut apelând procedura **Bar**, o elipsă umplută cu procedura **FillEllipse**. Aflați dumneavoastră singuri cum se folosesc aceste proceduri! Rulați exemplele din Help! Umplerile, adică haşurile, se stabilesc prin **SetFillStyle**. Aveți un exemplu mai jos. Încercați desenarea de histograme folosind Bar3D.

```
{$X+}
program DesenariDiverse;
uses Crt,Graph;
procedure OpenGraph;
var gd,gm: Integer;
begin gd := Detect; InitGraph(gd,gm,'C:\BP\BGI') end;
procedure CeFaceSetWriteMode;
var i: Byte;
begin
  SetWriteMode(0); {normal -desenare, desenare peste etc.}
  for i:=1 to 10 do
      begin
        Sound(100); Delay(10); NoSound; Line(100+10*i,100,100+10*i,150);
        Delay(100); Line(100+10*i,100, 100+10*i,150)
      end;
  SetWriteMode(1); {xor-desenare, stergere, desenare, stergere }
  for i:=1 to 10 do { acelasi lucru, dar alt efect => faceti animatie !! }
      begin
        Sound(400); Delay(10); NoSound; Line(100+10*i,200,100+10*i,250);
        Delay(100); Line(100+10*i,200,100+10*i,250)
      end;
  SetWriteMode(0) { revenire ! }
end;
procedure UtilizareFrumoasaRectangleSiCircle;
var i: Byte;
begin
  for i:=1 to 10 do
      begin
        SetColor(i); Rectangle(320-10*i, 240-10*i, 320+10*i,240+10*i);
        Circle(320,240,20*i); Sound(100+10*i); Delay(100)
      end;
  NoSound
end;
procedure CuBarTrasamHistograme;
const ceva=70;
var latime,lat2,n,i,valoare: Integer;
begin
  n := 10; ClearViewPort; { stergem ecranul, ca sa fie curat ! }
  latime := GetMaxX div (2*n); lat2 := latime div 2;
  for i:=1 to n do
      begin
        SetFillStyle(Random(11)+1, Random(10)+3);
        valoare:=Random(GetMaxY div 2);
        Bar(2*latime*i-lat2, valoare,2*latime*i+lat2, GetMaxY-ceva)
     end
end;
begin
  OpenGraph; CeFaceSetWriteMode; ReadKey;
  UtilizareFrumoasaRectangleSiCircle; ReadKey; CuBarTrasamHistograme;
  ReadKey; CloseGraph
end.
```

Afişări şi citiri în modul grafic

Afişarea textelor se face cu **`OutTextXY(x,y,sir)`**, care afişează şirul `sir`, începând din poziția `(x,y)`. Puteți schimba alinierea (implicit este la stânga şi sus) (prin `SetTextJustify`), precum şi mărimea caracterelor. Ba mai mult, puteți schimba chiar şi fontul cu care scrieți sau puteți scrie pe verticală! Folosiți, pentru aceasta, `SetTextStyle`.

Rulați exemplul de mai jos !

```
{$X+}
program AfisariSiCitiriInModGrafic;
uses Graph, Crt;
const ScriptFont=5; EuroFont=7;
procedure OpenGraph; { OG }
var gd,gm: Integer;
begin  gd:=Detect; InitGraph(gd,gm,'C:\BP\BGI')
end;
```

```
procedure AfiseazaSir(s: String; x,y,c,f,m: Integer); {AS }
begin
  SetTextJustify(LeftText,TopText); { alinierea stanga - sus }
  SetColor(c); SetTextStyle(f,HorizDir {scriere pe orizontala}, m);
  OutTextXY(x,y,s)
end;
```

```
procedure AfiseazaIntreg(v, x,y,c,f,m: Integer); { AI }
{afiseaza un numar intreg v, incepand cu punctul (x,y), de culoarea c,
in fontul f, de marimea m }
var sir: String[4];
begin { se converteste intregul v la un sir, afisabil cu OutTextXY }
  Str(v:4,sir); AfiseazaSir(sir,x,y,c,f,m)
end;
```

```
procedure AfiseazaReal(v: Real; x,y,c,f,m: Integer); { AR }
{ afiseaza un numar real v, incepand cu punctul (x,y), de culoarea c,
in fontul f, de marimea m }
var sir: String[7];
begin
     { se converteste intregul v la un sir, afisabil cu OutTextXY }
     Str(v:7:3,sir); { se afiseaza sirul corespunzator }
     AfiseazaSir(sir,x,y,c,f,m)
end;
```

```
procedure CitesteSir(x,y: Integer; mesaj: String; var S: String);
{ CS }
const Enter = #13; BackSpace = #8; Escape = #27;
      MultimeDeCaractereAcceptabile =
      ['0'..'9','A'..'Z','a'..'z','~','`','!','@','#','$','%',
      '^','&','*','(',')','_','-','+','=','\','|',',','.','/',
      '?','<','>',':',';','"',''', { apostroful insusi ! }
      '[',']','{','}',' ']; { spatiul }
var S_initial: String; tasta: Char;
begin
  S_initial:=S; SetTextStyle(DefaultFont,HorizDir,1); SetColor(White);
  SetTextJustify(LeftText,TopText); OutTextXY(x,y,mesaj+': _');
  x:=x+TextWidth(mesaj+': ');  S:='';
  repeat
    tasta:=ReadKey;
    if tasta=#0 then begin tasta:=ReadKey; Write(Chr(7)) end
    else
      if tasta in MultimeDeCaractereAcceptabile then
         begin
           { stergem fostul sir } SetColor(Black); OutTextXY(x,y,S+'_');
           S:=S+tasta; { afisam pe cel nou }
```

```
          SetColor(White); OutTextXY(x,y,S+'_')
        end
     else
        if tasta = BackSpace then
           if S='' then Write(#7) { sunet }
           else
             begin
               SetColor(Black); OutTextXY(x,y,S+'_'); SetColor(White);
               S:=Copy(S,1,Length(S)-1); OutTextXY(x,y,S+'_')
             end
        else
          if not (tasta in [Enter,Escape]) then Write(#7)
  until tasta in [Enter,Escape];
  if tasta = Escape then S:=S_initial;
  SetColor(Black);OutTextXY(x,y,S+'_');SetColor(White);OutTextXY(x,y,S)
end;
var s: String; n: Integer; x: Real; cod_eroare: Integer;
begin
  OpenGraph; { afisam un sir, un numar intreg si unul real }
  AfiseazaSir('AFISARI DIVERSE',100,20,LightBlue,ScriptFont,2);
  AfiseazaSir('siruri,numere intregi si reale',100,50,LightRed,EuroFont,3);
  CitesteSir(50,80,'Dati un numar intreg',s); Val(s,n,cod_eroare);
  if cod_eroare = 0 then AfiseazaIntreg(n,100,100,Yellow,SmallFont,5)
  else AfiseazaSir('Eroare !',100,90,LightGreen,SansSerifFont,1);
  CitesteSir(50,120,'Dati un numar real',s); Val(s,x,cod_eroare);
  if cod_eroare = 0 then AfiseazaReal(x,100,130,White,SmallFont,5)
  else AfiseazaSir('Eroare !',100,130,Cyan,SansSerifFont,1);
  SetTextJustify(CenterText,TopText);
  OutTextXY(GetMaxX div 2, 300,'Apasati o tasta !'); { scriere centrata }
  ReadKey; CloseGraph
end.
```

Procedurile din secțiunea anterioară sunt foarte utile în diferite programe de grafică, aşa că ar fi bine să le grupați într-o bibliotecă - `Graph1` - cu structura de mai jos. (Nu am rescris procedurile, ci am folosit două prescurtări pentru ele: `OG` pentru `OpenGraph` etc.). Pe parcursul cărții vom mai da unele proceduri importante, pe care va trebui să le includeți în acest unit, deoarece ele vor fi folosite şi în alte programe, mai complexe.

```
unit GRAPH1; { VEZI PROGRAMUL DE MAI SUS ! }
interface
    const ScriptFont=5; EuroFont=7;
    procedure OpenGraph;
    procedure AfiseazaSir(s: String; x,y,c,f,m: Integer);
    procedure AfiseazaIntreg(v, x,y,c,f,m: Integer);
    procedure AfiseazaReal(v: Real; x,y,c,f,m: Integer);
    procedure CitesteSir(x,y: Integer; mesaj: String;var S:String);
implementation
    uses Graph, Crt;
    procedure OpenGraph;
    OG
    procedure AfiseazaSir(s: String; x,y,c,f,m: Integer);
    AS
    procedure AfiseazaIntreg(v, x,y,c,f,m: Integer);
    AI
    procedure AfiseazaReal(v: Real; x,y,c,f,m: Integer);
    AR
    procedure CitesteSir(x,y: Integer; mesaj: String; var S: String);
    CS
end.
```

Uneori e necesar să operăm doar asupra unei zone de ecran. Acest lucru se realizează alegând un *viewport*, prin **`SetViewPort(x1,y1,x2,y2,clip)`**, unde `clip` este op valoare booleană, prin care se stabileşte dacă se doreşte decuparea imaginii la marginea zonei alese sau nu. Astfel, originea sistemului de coordonate devine `(x1,y1)`, orice afişare fiind relativă la acest punct. Dacă `clip` este `True`, atunci scrierile grafice nu vor depăşi zona stabilită, iar dacă este `False` vor putea depăşi. **Cu `ClearViewPort`** se şterge zona curentă (*viewport*-ul curent), iar cu **`ClearDevice`** totul ajunge ca la început, ecranul se şterge şi el.

```
{$X+}
program EfectDeSenila;
uses Graph, Crt;
procedure OpenGraph;
var gd,gm: Integer;
begin gd:=0; InitGraph(gd,gm,'C:\BP\BGI') end;
procedure SirMergator(s: String; x,y,numar, font, marime, cul: Integer);
{ plimba de la dreapta la stanga un sir "s", afisand cate "numar" caractere
din "s" simultan, in fontul "font" si de marimea "marime";
culoarea de afisare este "cul", iar prima pozitie de afisare este (x,y) }
const pauza=70;
var latime, inaltime,i: Integer;
begin { setam culoarea si stilul de scriere }
  SetColor(cul); SetTextStyle(font,HorizDir,marime);
  { punem spatii lui s }
  for i:=1 to numar do s:=' '+s+' ';
  { aflam latimea si inaltimea literelor, folosind litera cea mai mare }
  inaltime:=TextHeight('M'); latime:=TextWidth('M');
  {alegem o zona de ecran si o stergem}
  SetViewPort(x,y,x+latime*numar, y+inaltime+inaltime div 2, ClipOn);
  { ClipOn = True, ClipOff = False }
  ClearViewPort;
  { urmatorul dreptunghi este afisat in coordonate relative;
    el va incadra viewport-ul ales }
  Rectangle(0,0,latime*numar, inaltime+inaltime div 2);
  for i:=1 to Length(s)-numar do
      begin
        { afisarea e relativ la zona aleasa cu SetViewPort }
        OutTextXY(2,2, Copy(s,i,2*numar)); Delay(pauza); ClearViewPort;
        Rectangle(0,0,latime*numar, inaltime+inaltime div 2)
      end;
  ClearViewPort;
  { revenire la zona mare a ecranului, in coordonate absolute }
  SetViewPort(0,0,GetMaxX, GetMaxY,ClipOn)
end;
begin
  OpenGraph;
  SirMergator('Acest sir se plimba de la dreapta spre stanga !',
           150,200,20,TriplexFont, 2,Cyan);
  ReadKey; CloseGraph
end.
```

Am putea să scriem o procedură care să şteargă o anumită zonă de ecran, dată de coordonatele colțurilor stânga-sus şi dreapta-jos: `(x1,y1)` şi `(x2,y2)`. Includeți această procedură în unit-ul `GRAPH1`.

```
procedure ClearView(x1,y1,x2,y2: Integer);
begin
     SetViewPort(x1,y1,x2,y2,ClipOn); ClearViewPort;
     SetViewPort(0,0,GetMaxX,GetMaxY,ClipOn)
end;
```

Memorarea şi reafişarea imaginilor

Puteți stoca imagini undeva în memorie, apoi să le repuneți pe ecran. Următorul exemplu vă va arăta cum. Nu e nevoie să înțelegeți chiar tot ce este scris în acest program, deocamdată, ci cum să folosiți procedurile **GetImage**, **PutImage**. Restul îl veți înțelege pe parcurs, exersând.

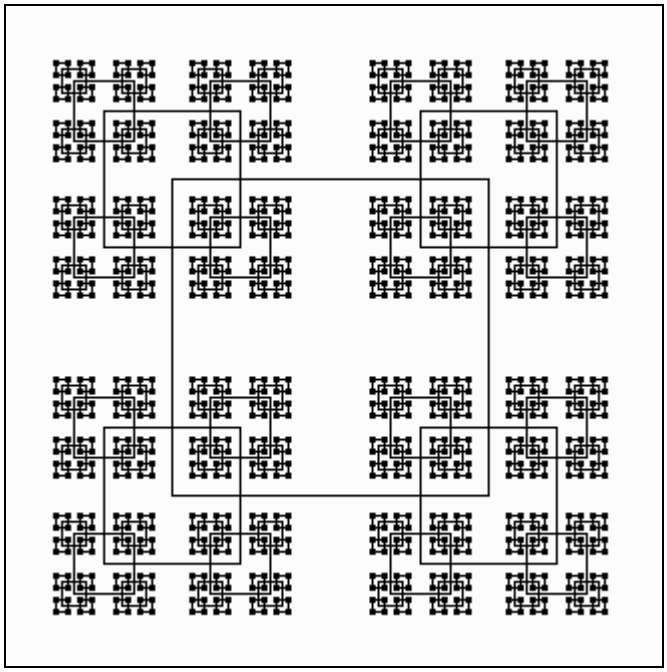
```
{$X+}
program Imagini;
uses Crt, Graph, Graph1;
procedure PreiaImagine(x1,y1,x2,y2: Integer; var Imaginea: Pointer; var
MarimeaImaginii: Word);
begin
  MarimeaImaginii := ImageSize(x1,y1,x2,y2);
  GetMem(Imaginea,MarimeaImaginii); GetImage(x1,y1,x2,y2,Imaginea^)
end;
var P: Pointer; Sz: Word;
begin
  OpenGraph; SetFillStyle(SolidFill,Green); FillEllipse(320,240,100,60);
  SetFillStyle(CloseDotFill, LightGreen); FillEllipse(300,220,50,20);
  PreiaImagine(300,200,400,300,P,Sz); Rectangle(300,200,400,300);
  ReadKey; ClearDevice; PutImage(100,100,P^,NormalPut);
  FreeMem(P,Sz); ReadKey; CloseGraph
end.
```

Imaginile se depun într-o zonă de memorie a cărei adresă de început este memorată într-o variabilă de un tip special, nou, numit **Pointer**. Înainte de a prelua imaginea de pe ecran (din zona determinată de coordonatele x1, y1, x2 şi y2), trebuie ca să se rezerve memorie pentru acest pointer. Acest lucru se face apelând mai întâi funcția **ImageSize**, care determină necesarul de octeți, apoi procedura **GetMem**, care are rolul lui New, dar pentru variabilele de tip Pointer. Preluarea propriu-zisă a imaginii este realizată de procedura **GetImage**. Afişarea imaginii se face cu **PutImage**, iar eliberarea zonei de memorie ocupată de aceasta cu procedura **FreeMem** (corespunzătoare lui Dispose). De reținut că nu pot fi stocate imagini prea mari. Numărul maxim de octeți ce pot fi ocupați de informațiigrafice este 65535.

Exersați, jucați-vă, distrați-vă! Vă dorim succes! Înainte de a trece la următoarea secțiune a lecției este bine ca singuri să vă puneți la punct procedurile grafice, exersând pe exemplele din Help-ul unit-ului Graph. Succes!

2. Figuri recursive

Să încercăm să desenăm urmărătoarea figură, pe care o vom numi "*diamant*".



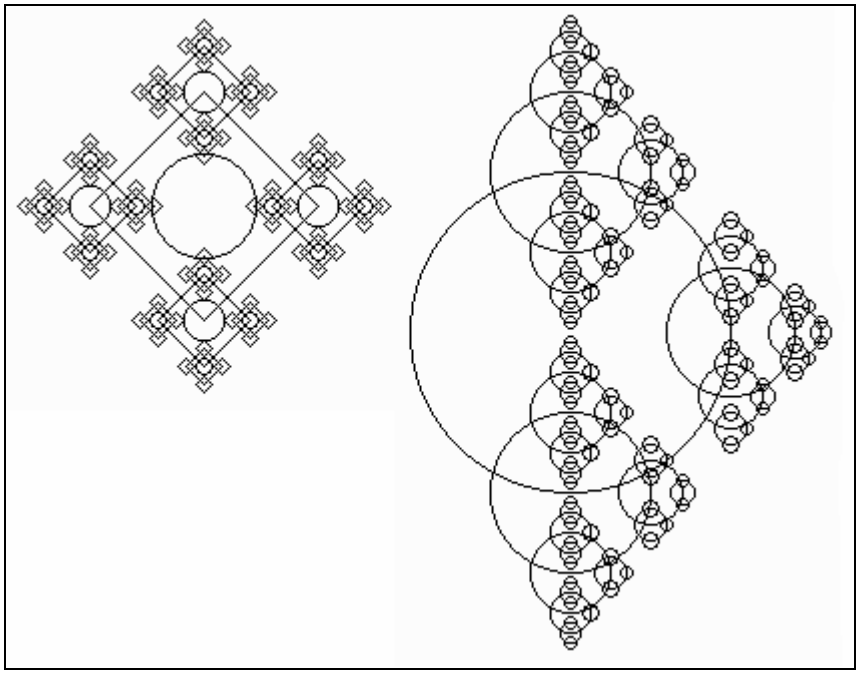
Observăm că un “diamant” este o figură recursivă. El este perfect caracterizat de coordonatele centrului său, precum şi de latura pătratului inițial. Pătratele imediat următoare, au laturile de c ori mai mici ş.a.m.d., unde c este o constantă reală oarecare, $c \geq 1$. De fapt, din ce constă un diamant? Dintr-un pătrat de centru `(x,y)` şi latură `l`, care are patru diamante în colțurile acestui pătrat. Aceasta este o definiție recursivă a diamantului. Ea ne va permite să scriem cu uşurință o procedură care să-l deseneze. Aceasta va desena un pătrat, apoi se va autoapela de patru ori, pentru cele patru diamante din colțuri. Însă nu putem merge aşa la infinit! De aceea, procedura va face efectiv desenări doar dacă latura pătratului este mai mare decât o anumită valoare minimă, de pildă `0`, `1` sau `2`.

Programul va fi, aşadar:

```
program FiguraRecursiva;
uses Graph, Graph1;
procedure Patrat(x,y,l: Integer);
var l2: Integer;
begin
  l2 := l div 2; Rectangle(x - l2, y - l2, x + l2, y + l2)
end;
{$S-}
procedure Diamant(x,y,l: Integer);
const c = 2.3;
var l2,l3: Integer;
begin
  if l>0 then
     begin
       Patrat(x,y,l); l2 := l div 2; l3 := Round(l/c);
       Diamant(x-l2,y-l2,l3); Diamant(x-l2,y+l2,l3);
       Diamant(x+l2,y-l2,l3); Diamant(x+l2,y+l2,l3)
     end
```

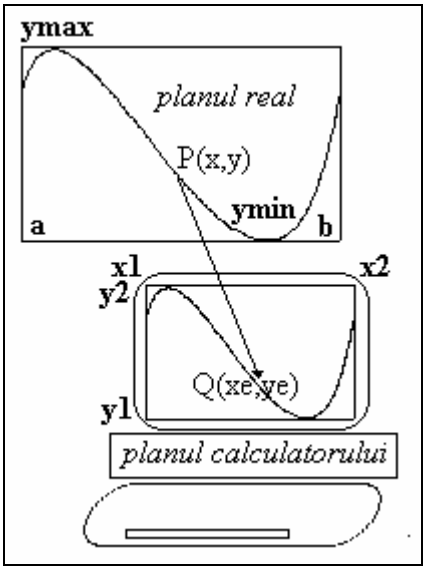
```
end;
{$S+}
begin
  OpenGraph; Diamant(GetMaxX div 2, GetMaxY div 2, GetMaxY div 3);
  ReadLn; CloseGraph
end.
```

Drept exerciţiu, propunem cititorului să realizeze, cu acceeaşi tehnică, următoarele desene



3. Graficul unei funcţii

Ne propunem, în continuare, să realizăm graficul unei funcţii oarecare `f:R→R` (definită în cadrul programului), într-o zonă dreptunghiulară `(x1,y1,x2,y2)`oarecare a ecranului, cu `y1>y2`. Funcţia va fi reprezentată pe un anumit interval `[a,b]`, deci $x \in [a,b]$. Se vor stabili, mai întâi, valorile minimă şi maximă ale funcţiei, pe intervalul `[a,b]`. Să zicem că ele sunt `ymin` şi `ymax`. Atunci vom face o transformare a imaginii reale a graficului în imaginea din zona de pe ecran a acestui grafic:



Această transformare va duce punctul `P(x,y)` din planul real în punctul `Q(xe,ye)` din planul calculatorului. De aceea vom utiliza două transformări liniare, care vor duce pe `x` în `xe`, iar pe `y` în `ye`. Funcțiile pe care le vom folosi vor avea forma `mx+n`. Pentru a determina - pentru ambele transformări - pe `m` şi `n`, va trebui să aplicăm formulele de transformare pentru capetele intervalelor. Aşadar, având: `xe=mx+n`, obținem `x1=ma+n`, respectiv `x2=mb+n`, de unde se determină `m` şi `n`, deci `xe` în funcție de `x`. La fel se determină `ye`, în funcție de `y`. În final, se vor obține formulele:

`xe=(x2-x1)*x/(b-a)+(b*x1-a*x2)/(b-a)` şi `ye=(y2-y1)*y/(ymax-ymin)+(ymax*y1-ymin*y2)/(ymax-ymin)`.

Valorile `xe` şi `ye` vor fi rotunjite la numere întregi, deoarece în ecranul grafic se lucrează cu coordonate întregi şi nu reale.

Iată mai jos un program - comentat - care realizează reprezentarea grafică a funcției `f(x)=x*sin(1/x)`, pt. `x` diferit de `0`, respectiv `0`, pentru `x=0`.

```
{$F+}  { optiune de compilare când se lucreaza cu variabile de tip functie,
de exemplu ca parametru, ca aici }
program ReprezentareGraficaFunctie;
uses Crt, Graph, Graph1;
type functie = function(x: Real): Real;
{ se defineste tipul functie cu argument real, care returneaza un real }
var x1,x2,y1,y2: Integer; { zona dreptunghiulara a ecranului, in care va fi
reprezentata grafic functia, cu autoscalare atât pe x, cât si pe y }
    a,b,   { capetele intervalului pe axa xx' }
    ymin,ymax: Real;  { extremele functiei pe axa yy' }
    sir: String; cod: Integer; { auxiliare }
{urmeaza functia care va fi reprezentata grafic; ea are tipul "functie" }
function f(x: Real): Real;
begin if x=0 then f:=0 else f:=x*sin(1/x) end;
procedure ReprezintaGrafic(f: functie; a,b: Real; x1,x2,y1,y2: Integer);
var x0,y0, {sunt coordonatele pentru axele xx', yy', deci pt. O(o,o)}
      xe,ye,i: Integer; {xe,ye-coordonate ecran, i=contor pt. reprezentare}
      xev,yev, aux: Integer; { coordonate ecran vechi }
      x,y: Real; { coordonatele din planul real }
begin   { se traseaza zona dreptunghiulara pt. reprezentarea functiei }
  Rectangle(x1,y1,x2,y2);
  ymin:=f(a);
 {se determina ymin si ymax, parcurgând [a,b] si comparând valorile curente
  stabilite, cu valorile functiei in fiecare punct considerat }
  ymax:=ymin;
  for i:=0 to (x2-x1) do
      begin
        x:=a+i*(b-a)/(x2-x1); { punctul curent considerat }
        y:=f(x); { valoarea in punctul curent }
        if y>ymax then ymax:=y; { actualizare y maxim }
        if y<ymin then ymin:=y   { actualizare y minim }
     end;
  x0 := Round((b*x1-a*x2)/(b-a)); { se determina x origine }
```

```
  { se va trasa axa yy' doar daca x0 este in intervalul [a,b] }
  if (a<=0) and (0<=b) then Line(x0,y2,x0,y1);
  y0 := Round((ymax*y1-ymin*y2)/(ymax-ymin)); { la fel pt. y0 si axa xx' }
  if (ymin<=0) and (0<=ymax) then Line(x1,y0,x2,y0);
  { initializarea coordonatelor ecran vechi }
  x := a; y := f(a);
  xev := Round((x2-x1)*x/(b-a)+(b*x1-a*x2)/(b-a));
  yev := Round((y2-y1)*y/(ymax-ymin)+(ymax*y1-ymin*y2)/(ymax-ymin));
  for i:= 0 to (x2-x1) do
   begin
    x:=a+i*(b-a)/(x2-x1); { stabilirea a (x2-x1+1) puncte curente x }
    xe:=Round((x2-x1)*x/(b-a)+(b*x1-a*x2)/(b-a));{ transf. x in x ecran }
    y:=f(x); { determinarea valorii reale a functiei f in punctul x }
    ye:=Round((y2-y1)*y/(ymax-ymin)+(ymax*y1-ymin*y2)/(ymax-ymin));
    Line(xev,yev,xe,ye); { trasarea unei linii de la fostul Q la Q(xe,ye)}
    xev := xe; yev := ye { actualizarea valorilor "vechi" }
   end
end;
begin
  OpenGraph;
  AfiseazaSir('Graficul functiei 0, daca x=0, respectiv x*sin(1/x), altfel',
              10,10,White,EuroFont,1);
  CitesteSir(50,70,'Dati a = ',sir); Val(sir,a,cod);
  CitesteSir(500,70,'Dati b = ',sir); Val(sir,b,cod);
  ReprezintaGrafic(f,a,b,50,600,400,100); ReadLn; CloseGraph
end.
```

Fireşte, dacă ne propunem să realizăm o trasare a graficului funcţiei prin puncte şi nu prin linii, va trebui să nu mai memorăm fostele coordonate ecran (`x` ecran vechi, `y` ecran vechi: `xev`, `yev`), şi apoi să trasăm o linie `Line(xev,yev,xe,ye)`, ci să punem simplu `PutPixel(xe,ye,GetColor)`.

Propunem cititorului să combine programul anterior cu unul care să verifice corectitudinea sintactică a unei funcţii, precum şi cu evaluatorul de expresii prezentat în lecţia 5.
Astfel, cititorul va putea să realizeze un program care să reprezinte grafic orice funcţie, a cărei expresie să fie dată de la tastatură. În loc de funcţia `f` din programul anterior, ar apare funcţia `ValoareFunctie` din programul din lecţia 5.
În privinţa verificării corectitudinii sintactice a expresie funcţiei, se va folosi faptul că după un caracter de un anumit tip nu poate urma decât caractere de anumite tipuri, prin *tipuri* înţelegând valorile returnate de funcţia `TipCaracter` din secţiunea respectivă a lecţiei 5.
Astfel, de pildă, după un caracter de tipul `3` (paranteză rotundă deschisă) nu poate urma decât un caracter de unul din tipurile: `3`, `5`(**`-`**), `6` (**`+`**), `10` (variabila `x`), `2` (început de funcţie) sau `1` (început de număr). Nu vom avea niciodată ceva de genul `(*` sau `()` sau `(^`. Lăsăm ca exerciţiu cititorului să realizeze o funcţie care să verifice dacă o expresie poate reprezenta o funcţie sau nu. Dacă nu să se semnaleze eroare.
De asemenea, ar fi bine să se verifice dacă funcţia se poate evalua într-un anumit punct, deci dacă funcţia are domeniul de definiţie bine dat. Dacă nu, să se semnaleze eroare.
Toate aceste probleme îşi găsesc rezolvarea în lucrarea mea "*Grafică în OOP ... şi nu numai ...*", Editura *Adias*. În respectiva carte este prezentat textul sursă al unui program orientat pe obiecte, care permite reprezentarea grafică a mai multor funcţii, oricât de complexe.

4. Pictură cu mouse-ul

În lecția a şasea am vorbit despre cum se poate utiliza mouse-ul în modul text. În lecția a zecea am prezentat o bibliotecă UTMouse care să permită folosirea procedurilor de mouse în modul text. Una asemănătoare, dar pentru modul grafic, este dată mai jos. Ea se numeşte uMouse şi va fi folosită pentru a realiza un program de desenare simplificat, de genul lui PaintBrush.

Mouse-ul în modul grafic

Procedurile din biblioteca uMouse sunt aceleaşi ca şi în cazul modului text, cu excepția celei ce determină butonul apăsat şi poziția cursorului de mouse pe ecran. Cursorul de mouse în modul grafic este o săgeată. Noua procedură se numeşte MouseData. Biblioteca uMouse cuprinde şi o procedură nouă MouseMove, care poziționează săgeata mouse-ului într-un anumit punct.

```
unit uMouse; { UMOUSE.PAS }
interface
         procedure MouseInit;
         procedure MouseShow;
         procedure MouseHide;
         procedure MouseData(var buton,x,y: Integer);
         procedure MouseMove(x,y: Integer);
implementation
    procedure MouseInit; assembler;
    asm mov ax,0; int $33 end;
    procedure MouseShow; assembler;
    asm mov ax,1; int $33 end;
    procedure MouseHide; assembler;
    asm mov ax,2; int $33 end;
    procedure MouseData(var buton, x, y: Integer);
    var bb,xx,yy: Integer;
    begin
      asm
        mov ax,3; int $33; mov bb,bx; mov xx,cx; mov yy,dx
      end;
      buton:=bb; x:=xx; y:=yy
    end;
    procedure MouseMove(x,y: Integer); assembler;
    asm MOV AX,4; MOV CX,X; MOV DX,Y end;
end.
```

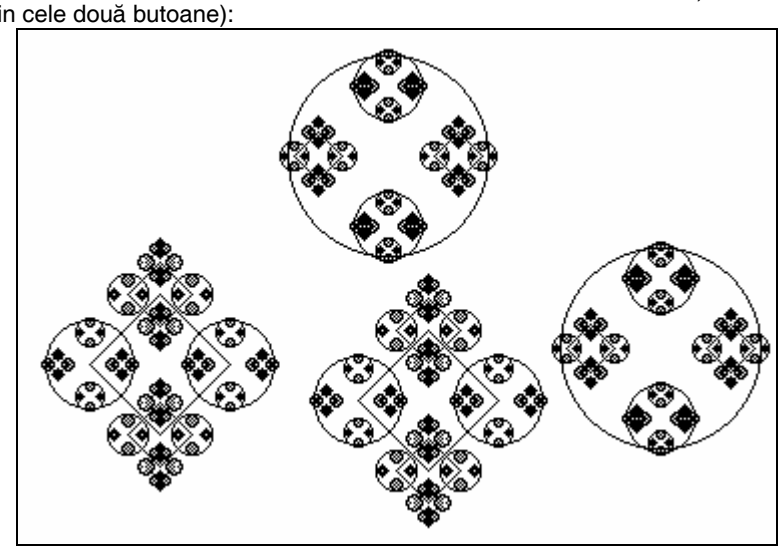
Următorul program realizează un desen foarte frumos, format din “diamante” (vezi secțiunea a doua a lecției!). Cele două diamante sunt desenate de două proceduri recursive. Fiecare o apelează pe cealalta, deci avem o recursivitate încrucişată.

Desenarea fiecăruia din cele două diamante se realizează la apăsarea butonului corespunzător al mouse-ului. Dacă se acționează ambele butoane simultan sau o tastă, atunci programul se opreşte.

```
program DesenareDiamanteCuMouse;
uses Crt, Graph, Graph1, uMouse;
const nici_un_buton=0;buton_stanga=1; buton_dreapta=2; ambele_butoane=3;
procedure Romb(x,y,l: Integer);
var l2: Integer;
begin
  l2 := Round(l/Sqrt(2)); Line(x-l2, y, x, y-l2);
  Line(x, y-l2, x+l2, y); Line(x+l2, y, x, y+l2); Line(x, y+l2, x-l2, y)
end;
procedure DiamantCercuri(x,y,l:Integer);forward;{recursivitate incrucisata!}
{$S-}
procedure DiamantRomburi(x,y,l: Integer);
const c = 2.3;
```

```
var l2,l3: Integer;
begin
  if l>0 then
     begin
       Romb(x,y,l); l2 := Round(l/Sqrt(2)); l3 := Round(l/c);
       DiamantCercuri(x-l2,y,l3); DiamantRomburi(x,y-l2,l3);
       DiamantCercuri(x+l2,y,l3); DiamantRomburi(x,y+l2,l3)
     end
end;
procedure DiamantCercuri(x,y,l: Integer);
const c = 3;
var l2,l3: Integer;
begin
  if l>0 then
     begin
       Circle(x,y,l); l2 := Round(l/Sqrt(2)); l3 := Round(l/c);
       DiamantRomburi(x-l2,y,l3); DiamantCercuri(x,y-l2,l3);
       DiamantRomburi(x+l2,y,l3); DiamantCercuri(x,y+l2,l3)
     end
end;
{$S+}
var buton,x,y: Integer;
begin
  OpenGraph; MouseInit; MouseShow;
  repeat
    MouseData(buton,x,y);
    case buton of
      buton_stanga: begin MouseHide; DiamantCercuri(x,y,50); MouseShow end;
      buton_dreapta: begin MouseHide; DiamantRomburi(x,y,50); MouseShow end
    end
  until (buton=ambele_butoane) or KeyPressed; CloseGraph
end.
```

Un exemplu de figură realizată cu acest program este următorul (s-a acționat de două ori fiecare din cele două butoane):



Programul "*Desenator*"

Programul următor reprezintă un editor grafic foarte simplu, ce permite desenarea de linii şi dreptunghiuri în mod interactiv, cu ajutorul mouse-ului. Comenzile sunt preluate din partea de sus a ecranului, unde se află un meniu.

Desenarea liniilor şi a dreptunghiurilor se va face în maniera cunoscută din *PaintBrush*. Se selectează primul punct, iar cât timp nu s-a dat drumul butonului de mouse, se desenează linia (dreptunghiul) între primul punct şi punctul curent, de două ori, în modul de scriere *xor*, ca a doua oară să aibă loc, de fapt, o ştergere. Când se dă drumul la buton, atunci se va trasa linia finală, ce trebuie să se încadreze în zona de desenare. Prin urmare, procedurile de desenare linii şi dreptunghiuri vor funcționa la fel.

Comanda selectată din meniu va fi afişată în altă culoare. Structura programului este foarte generală şi am realizat-o cât mai clar, pentru ca cititorul să poată extinde programul după preferințe, aducându-i noi facilități. Fireşte, în primul rând, ar fi bine ca să se realizeze salvarea de imagini pe disc, restaurarea lor. În acest sens, se vor folosi procedurile `GetImage` şi `BlockWrite` (pentru salvare), respectiv `BlockRead` şi `PutImage` pentru restaurare pe ecran. Aceste proceduri vor salva imaginea în formatul grafic specific mediilor *Borland*. O altă posibilitate ar fi utilizarea unui format grafic clasic, cum ar fi formatul `BMP`, despre care puteți afla detalii din lucrarea *Grafică în OOP ... şi nu numai ...* .

```
program Desenator;
uses Crt, Graph, Graph1, uMouse;
const inaltime=15;
const NrCom=3; cmNimic=0; cmLinie=1; cmDreptunghi=2; cmExit=NrCom;
const NumeCom: array[1..NrCom] of String=('Linie','Dreptunghi','Stop');
var Comanda, ComandaVeche: Byte;
function Apartine(x,y, x1,y1,x2,y2: Integer): Boolean;
begin Apartine := (x1<=x) and (x<=x2)and (y1<=y) and (y<=y2) end;
procedure Meniu;
var i: Byte; lat: Integer;
begin
     MouseHide; lat:=GetMaxX div NrCom;
     SetTextJustify(CenterText, CenterText);
     for i:=1 to NrCom do
         begin
           if i = Comanda then SetColor(Yellow) else SetColor(White);
           Rectangle(lat*(i-1),0,lat*i-1,inaltime);
           OutTextXY(lat*(i-1) + lat div 2,inaltime div 2, NumeCom[i])
         end;
     MouseShow
end;
procedure StabilesteComanda;
var i,b,x,y, lat: Integer;
begin
  lat := GetMaxX div NrCom; MouseData(b,x,y);
  if b=1 then
     for i:=1 to NrCom do
         if (Apartine(x,y,lat*(i-1),0,lat*i,inaltime)) or
            (Apartine(x,y,lat*(i-1),inaltime,lat*i,inaltime) )
            then Comanda := i
end;
procedure DesLinie;
var b,px,py,x,y: Integer;
begin
  SetColor(White); MouseData(b,px,py);
  if (Apartine(px,py,0,inaltime,GetMaxX,GetMaxY)) and (b=1) then
     begin
       SetWriteMode(XorPut); MouseHide;
       repeat
         MouseData(b,x,y);
```

```
        if y<inaltime+1 then
           begin y:=inaltime+1; MouseMove(x,inaltime+1) end;
        Line(px,py,x,y); MouseShow; Delay(50); MouseHide; Line(px,py,x,y)
      until b=0;
      SetWriteMode(NormalPut); Line(x,y,px,py); MouseShow
    end
end;
procedure DesDreptunghi;
var b,px,py,x,y: Integer;
begin
  SetColor(White); MouseData(b,px,py);
  if Apartine(px,py,0, inaltime,GetMaxX,GetMaxY) and (b=1) then
     begin
       SetWriteMode(XorPut); MouseHide;
       repeat
         MouseData(b,x,y);
         if y<inaltime+1 then
            begin y:=inaltime+1; MouseMove(x,inaltime+1) end;
         Rectangle(px,py,x,y); MouseShow; Delay(50); MouseHide;
         Rectangle(px,py,x,y)
       until b=0;
       SetWriteMode(NormalPut); Rectangle(x,y,px,py); MouseShow
     end
end;
procedure ExecutaComanda;
begin
  case Comanda of cmLinie: DesLinie; cmDreptunghi: DesDreptunghi end
end;
begin
  OpenGraph; MouseInit; MouseShow;
  Comanda:=cmNimic; ComandaVeche:=cmNimic; Meniu;
  repeat
    StabilesteComanda;
    if (Comanda <> ComandaVeche) and (Comanda <> cmNimic) then
       begin Meniu; ComandaVeche:=Comanda end;
    ExecutaComanda
  until (Comanda = cmExit) or KeyPressed;
  CloseGraph
end.
```

5. Exploatarea fişierelor de caractere CHR

Avem convingerea că acei dintre dumneavoastră care au realizat programe de grafică în Turbo-Pascal au folosit de multe ori procedurile referitoare la fonturi, adică la seturile de caractere care stau la baza afişării de texte în modul grafic. Prin `SetTextStyle`, se poate alege un anumit font, o anumită direcţie (*orizontal* sau *vertical*) şi o anumită mărime a caracterelor care vor fi afişate ulterior cu `OutText` sau `OutTextXY`. Dar v-aţi pus vreodată problema cum am putea afişa un text pe oblică? Sau sub forma unui cerc? Dacă da, probabil că doriţi şi un răspuns la astfel de întrebări. Această secţiune a lecţiei încearcă să vă răspundă la astfel de probleme.

Fireşte, scrierea unui text pe oblică nu este posibilă, oricât şi oricum am utiliza procedurile standard ale bibliotecii de grafică `Graph`, aşa că va trebui să vedem cum sunt codificate fonturile în fişierele `CHR`. Ele sunt nişte fişiere cu un anumit format, având un *antet* (“*header*”), urmate de o serie de date pentru fiecare caracter în parte, care îl descriu sub forma unor linii.

Aşadar, pentru a putea rezolva problema afişării sub un anumit unghi a unui caracter, va trebui să rezolvăm următoarele patru subprobleme:

- să decodificăm fişierul cu fontul de caractere pe care dorim să îl folosim;
- să vedem care este reprezentarea internă a unui caracter din fontul de caractere ales;
- să cunoaştem o formulă de rotire a unui punct în jurul altui punct cu un unghi dat;
- să redesenăm toate liniile caracterului care se va afişa, în conformitate cu rotirea executată.

În privința formulelor matematice care ne dau noile coordonatele ale unui punct, rotit cu un anumit unghi față de alt punct, acestea pot fi obținute din cărți de geometrie analitică, sau chiar din procedura ce le va folosi. În cele ce urmează vom prezenta formatul fişierelor `CHR`, apoi, în următoarea secțiune vom da textul sursă al unui program de testare, care conține o procedură de afişare pe oblică a unui şir de caractere.

Formatul fişierelor de caractere `CHR`

Aceste fonturi, numite şi "încondeiate" ("*stroke fonts*", în engleză) definesc caracterele ca o secvență de linii (trăsături), în opoziție cu fonturile de tip *bitmap*, care definesc caracterele ca o matrice de puncte.

Avantajul fonturilor încondeiate este acela că ele sunt vectoriale, deci pot fi scalate la mărimi arbitrare şi pot să-şi mențină rezoluția lor. Fonturile de tip *bitmap* (hartă de biți, punct cu punct) sunt făcute pentru o anumită mărime a punctului şi nu pot fi scalate prea bine. De exemplu, dacă aveți un font de tip *bitmap* pentru o matrice de 72 puncte pe inch (DPI) şi mărim de patru ori în înălțime şi în lățime punctele pentru a utiliza o imprimantă laser cu 300 DPI, atunci pot apărea margini prea mari (largi) în fontul 72 DPI.

Pe de altă parte, fonturile de tip stroke, nu sunt convertite în puncte, pînă cînd rezoluția perifericului de ieşire nu este cunoscută. Astfel, dacă vreți să scrieți cu fonturi de tip *stroke* pe o imprimantă laser cu 300 DPI, punctele care trasează liniile caracterelor vor fi tipărite la 300 DPI.

Acestea, fiind spuse, să trecem la prezentarea structurii unui fişier de tip *CHR*, exemplificînd pe fişierul `TRIP.CHR`, care este acelaşi pentru toate mediile de programare *Borland*.

```
;  offset 0h este un header specific firmei Borland:
      HeaderSize        de ex.   080h
      DataSize          de ex.   (mărimea fişierului)     ; lungimea fis. CHR
      descr             de ex.   "Triplex font"           ; descriere
      fname             de ex.   "TRIP"                   ; numele fisierului
      MajorVersion   de ex.   1                           ; versiunea fontului
      MinorVersion   de ex.   0                           ; subversiunea
      db    'PK',8,8
      db    'BGI ',descr,'  V'
      db    MajorVersion+'0'
      db    (MinorVersion / 10)+'0',(MinorVersion mod 10)+'0'
      db    ' - 19 October 1987',0DH,0AH
      db    'Copyright (c) 1987 Borland International', 0dh,0ah        ; (c)
      db    0,1ah                                       ; null & ctrl-Z = end
      dw    HeaderSize                                            ; mărimea header-ului
      db    fname                                       ; numele fontului
      dw    DataSize                                    ; mărimea fis. font
      db    MajorVersion,MinorVersion                   ; numărul de versiune
      db    1,0                                         ; nre. de versiune min.
      db    (HeaderSize - $) DUP (0)                    ; tampon
```

La offset-ul 80h încep datele pentru fişier, după cum urmează:

```
;       80h    '+'     indicatoare pentru tipul fisierului stroke
;       81h-82h        numărul de caractere în fisierul font (n)
;       83h                    nedefinit
;       84h            valoarea ASCII a primului caracter din fisier
;       85h-86h        offset pt. definitiile stroke (8+3n)
;       87h            scan flag (normal 0)
;       88h            distanta de la origine la vîrful literei mari
;       89h            distanta de la origine la linia de bază
;       90h            distanta de la origine la cea mai coborâtă linie
;       91h-95h        nedefinit
;       96h            offset-uri pentru definitii individuale de caractere
;       96h+2n         tabela grosimilor (un word pentru fiecare caracter)
;       96h+3n         startul definitiilor caracter
```

Definițiile individuale de caractere constau dintr-un număr variabil de cuvinte (word) (16 bits), descriind operația necesară în desenarea caracterului. Fiecare word constă dintr-o pereche de coordonate `(x,y)` şi două coduri de operații pe doi biți.

Cuvintele sunt codificate după cum urmează:

```
Byte 1        7  6  5  4  3  2  1  0    bit #
  op1  <coord. X pe 7 biti cu semn>
Byte 2        7  6  5  4  3  2  1  0    bit #
  op2  <coord. Y pe 7 biti cu semn>
```

Codurile de operații sunt:

```
op1=0  op2=0  Sfârsitul definitiei de caracter.
op1=1  op2=0  Mută pointerul grafic în punctul de coordonate (x,y).
op1=1  op2=1  Trasează o linie de la punctul curent la punctul de coordonate (x,y).
```

Formatul fişierelor `CHR` este scris în unit-ul `uFont` dat mai jos, folosind structuri din limbajul Turbo-Pascal.

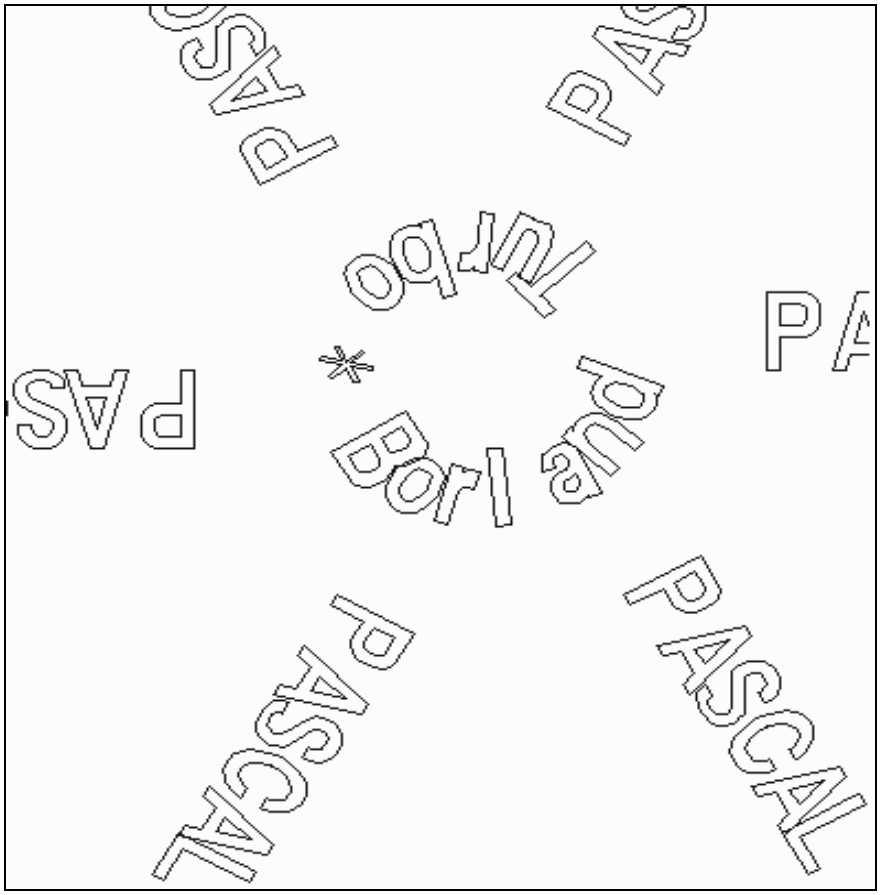
```
{ Informatiile din header-ul unui fisier de fonturi CHR }
unit uFont;
interface
const FOREVER=True; Prefix_Size=128;
      {0x80} Major_Version=1;
      Minor_Version=0; SIGNATURE='+';
type
  OP_CODES = (END_OF_CHAR, DO_SCAN, MOVE, DRAW);
  THEADER = record
             sig: Char; { SIGNATURE byte }
             nchrs: Integer; { numarul de caractere in fisier }
             mystery: Char; { nedefinit inca }
             first: Char; { primul caracter din fisier }
             cdefs: Integer; { offset la definitiile de caractere }
             scan_flag: ShortInt; {1 daca setul de caractere este scanabil}
             org_to_cap: ShortInt; {inalt. de la orig. la vf literei mari}
             org_to_base: ShortInt; {inalt. de la origine la linia de baza}
             org_to_dec: ShortInt; {inalt. de la origine la linia infer.}
             fntname:array[0..3] of Char; {4 caractere pt. numele fontului}
             unused: Char {nedefinit inca}
            end;
  STROKE=record
```

```
           opcode: ShortInt; { Byte pt. codul de operatie }
           x,y: Integer; { Offset relativ pe directia x, y }
         end;
  PSTROKE=^STROKE;
  FHEADER=record
           header_size: Word; {Versiunea 2.0 a formatului de header}
           font_name: array[0..3] of Char; {Numele intern al fontului}
           font_size: Word; {Marimea in bytes a fisierului}
           font_major,font_minor:Byte; {Informatii de versiune driver}
           min_major,min_minor: Byte; {Informatii de revizie pentru BGI}
         end;
implementation   end.
```

Proceduri de afişare speciale

Programul demonstrativ care urmează conţine o procedură speciale de afişare a unui şir de caractere pe oblică, care nu putea fi realizată prin metode clasice. Ideea care stă la baza ei este de a roti liniile care compun caracterele ce formează şirul de afişat, în conformitate cu fontul utilizat. Programul realizează o încărcare a fontului dintr-un fişier cu extensia `CHR` de pe disc. Propunem cititorului să îmbunătăţească programul pentru a putea folosi mai multe fonturi, pe rând, pentru diferite texte.

În program sunt utilizate, printre altele, tipul de date `Pointer`, care reprezintă un *pointer* către o zonă de memorie oarecare, care poate conţine date de orice gen. (Până acum tipul `Pointer` a fost utilizat pentru a stoca o imagine grafică.). De asemenea, avem de a face cu operaţii asupra pointer-ilor mai puţin cunoscute, cum ar fi, de pildă, incrementarea unui pointer `p`, `Inc(p)`, care înseamnă că `p` va referi următoarea dată din zona respectivă de memorie. Detalii se pot obţine din `Help`-ul mediului *Borland Pascal*, pentru cititorii preocupaţi de grafică contează mai mult cum să se utilizeze aceste proceduri şi mai puţin cum sunt ele efectiv construite. Mai jos este dat rezultatul programului şi textul său sursă.




```
{$X+}
program TestFonturiCHR;
uses Crt, Graph, Graph1, uFont;
var c: Char; ffile: File; Font: PChar;
{ Pointer la zona de stocare a font-ului }
  Prefix: array[0..Prefix_Size-1] of Char;
{ File Prefix Holder }
  Header: THEADER;
{ File Data Header }
  Offset: array[0..255] of Integer;
{ Font data offsets }
  Char_Width: array[Char] of ShortInt;
{ Character Width Table }
  i: Integer;
const POZX: Integer = 25; POZY: Integer = 25;
type WordPtr = ^Word;
var lungime, current: LongInt; cptr: PChar;
{ Decodifica un sigur "cuvant" din fisier,
obtinand o inregistrare de tip "stroke". }
function Decode(iptr: Word; var x,y: Integer): Integer;
type TDECODE = record {7,1,7,1}
                 xoff: Integer;
```

```
                flag1: Word; yoff: Integer; flag2: Word
              end;
var cword:TDECODE;unu,doi:Byte;dec:Integer; {redefinire identificator!}
begin
  unu := Lo(iptr); doi := Hi(iptr);
  if unu >= 128 then
     begin
       cword.flag1 := 1; unu:=unu-128;
       if unu>=64 then cword.xoff:=63-unu else cword.xoff := unu
     end
  else
    begin
      cword.flag1 := 0;
      if unu>=64 then cword.xoff:=63-unu else cword.xoff:=unu
    end;
  if doi >= 128 then
     begin
       cword.flag2 := 1; doi := doi - 128;
       if doi>=64 then cword.yoff:=63-doi else cword.yoff := doi
     end
  else
    begin
      cword.flag2 := 0;
      if doi>=64 then cword.yoff:=63-doi else cword.yoff := doi
    end;
  x := cword.xoff; y := cword.yoff;
  if y < Header.org_to_dec then y:=Header.org_to_dec;
  if y > Header.org_to_cap then y:=Header.org_to_cap;
  dec:=(cword.flag1 shl 1)+cword.flag2; Decode := dec
end;
{Decodifica formatul fisierului intr-o struct. interna mai usor de folosit}
function UnPack(buf: PChar;index: Integer; var neww: PSTROKE): Integer;
var pb: WordPtr; po: PSTROKE; num_ops,jx, jy, opcode, i, opc: Integer;
begin
  num_ops:=0; New(pb); pb:=WordPtr(buf+index);
  repeat{Pt. fiecare byte din buffer}
    Inc(num_ops); {Contorizeaza operatie}
    opcode := Decode(pb^, jx, jy ); {Decodifica inregistrarea de date}
    Inc(pb);
  until opcode = 0;
  GetMem(neww,num_ops*SizeOf(STROKE)); GetMem(po,num_ops*SizeOf(STROKE));
  po := neww; pb := WordPtr(buf + index); {Pune pointer la inc buffer-ului}
  for i:=1 to num_ops do { Pt. fiecare cod de operatie din buffer }
    begin
      opc := Decode(pb^, po^.x, po^.y);{Decodifica campul de date}
      Inc(pb); {se trece la urmatoarea locatie de memorie}
      po^.opcode := opc; {Salveaza codul de operatie opcode}
      Inc(po)
    end;
  UnPack := num_ops {returneaza contorul de operatii}
end;
procedure WriteChar(litera: char; alfa: Real; x0, y0: Integer);
var sptr: PSTROKE; j, i, xx, yy, xxr, yyr, Caracter: Integer;
begin
 i := Ord(litera)-Ord(Header.first);
 Caracter:=UnPack(Font, Offset[i], sptr); y0 := GetMaxY - y0;
 for j:=1 to Caracter do
     begin
       xx := x0+sptr^.x; yy := y0+sptr^.y;
       xxr := Round((xx-x0)*cos(alfa) - (yy-y0)*sin(alfa) + x0);
       yyr := Round((xx-x0)*sin(alfa) + (yy-y0)*cos(alfa) + y0);
       if sptr^.opcode = 2 then MoveTo(xxr, GetMaxY-yyr);
```

```
       if sptr^.opcode = 3 then LineTo(xxr, GetMaxY-yyr);
       Inc(sptr)
   end
end;
procedure WriteStr(cuvint: String; alfa: Real; x0, y0: Integer);
var l,i: Byte;
begin
  POZX := x0; POZY := y0; l := Length(cuvint);
  for i:=1 to l do
      begin
        WriteChar(cuvint[i], alfa, POZX, POZY); Inc(POZX, Round (
        Char_Width[cuvint[i]]*cos(alfa))); Dec(POZY,
        Round(Char_Width[cuvint[i]]*sin(alfa))); Inc(x0, Round(
        Char_Width[cuvint[i]]*cos(alfa)));
        Dec(y0, Round(Char_Width[cuvint[i]]*sin(alfa)))
      end
end;
procedure CircleStr(cuvint: String; x0, y0, raza: Integer);
var alfa: Real; l,i: Byte; xx,yy: Integer;
begin
  l := Length(cuvint); y0 := GetMaxY - y0;
  for i:=1 to l do
      begin
        alfa := -2*pi*i/l;
        xx:=Round(x0+raza*cos(alfa)); yy:=Round(GetMaxY-y0+
        raza* sin(alfa)); WriteChar(cuvint[i], pi/2-alfa, xx, yy)
      end
end;
begin { program }
  Assign(ffile,'BOLD.CHR'); {$I-}Reset(ffile,1){$I+};
  if IOResult <> 0 then Halt(1);
  BlockRead(ffile, Prefix, SizeOf(Prefix));
  cptr := Prefix;  { incepe la baza prefixului }
  while cptr^<> #26 do Inc(cptr);
  cptr := #0; { Taie prefixul la caracterul EOF }
  BlockRead(ffile,Header,SizeOf(THEADER));
  BlockRead(ffile, Offset,Header.nchrs*SizeOf(Integer));
  BlockRead(ffile, Char_Width,Header.nchrs*SizeOf(Char));
  current := FilePos(ffile);lungime := FileSize(ffile)-current;
  { incarca datele despre font }
  if MaxAvail<lungime then
     begin
       Close(ffile); WriteLn('Memorie insuficienta !'); Halt(1)
     end;
  GetMem(Font, lungime); { creaza spatiu pentru date }
  BlockRead(ffile, Font^, lungime); { incarca } Close(ffile);
  OpenGraph;
  Rectangle(0, 0, GetMaxX, GetMaxY);
  for i:=-3 to 2 do
      begin
        SetColor(i);
        if i=0 then SetColor(Cyan);
        WriteStr('     PASCAL',-pi*i/3,320,200);
      end;
  SetColor(White);
  CircleStr(' Turbo * Borland ',320,200,80);
  ReadKey; FreeMem(Font,lungime); CloseGraph
end.
```

6. Animație. Fişierele FLI

Tehnici simple de animație

Ca **tehnici simple de realizare a animației** în modul grafic menționăm:

- ştergerea figurii animate din vechea poziție, apoi redesenarea în noua poziție a sa; se poate face o ştergere din vechea poziție fie folosind culoarea fondului, fie prin rescriere în modul `XorPut`, ales cu `SetWtiteMode`; o asemenea tehnică este folosită în primul program de animație prezentat mai jos;
- cel de al doilea program foloseşte procedurile de lucru cu imagini, `PutImage` şi `GetImage`;
- un alt procedeu este cel al alternării paginilor video, de pildă în modul `EGA`, aşa cum se va folosi în programul din secțiunea următoare.

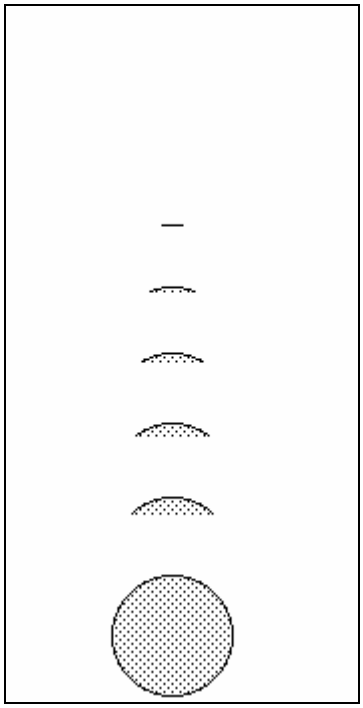
Programul `PingPong1` prezintă un cerc în mişcare, de fapt o bilă de ping-pong în cădere liberă. Ea execută câteva salturi, din ce în ce mai mici în înălțime, până se opreşte de tot pe "masă". Cercul se desenează în fiecare poziție de două ori, mai întâi cu culoarea bilei, apoi cu culoarea `0`, a fondului; a doua desenare va determina ştergerea pixelilor ce compun figura. Procedeul nu ar fi prea eficient în cazul în care s-ar cere ca să avem un cerc haşurat, deoarece ar fi cam încet şi s-ar observa unele pâlpâiri foarte deranjante ale imaginii pe ecran.

```
{$X+}
program PingPong1;
{ stergere cu negru }
uses Crt, Graph, Graph1;
procedure Sunet;
begin
  Sound(200); Delay(100); NoSound
end;
const variatia_inaltimii=50; variatia_acceleratiei=1; marimea=20; pauza=40;
      culoarea=Yellow; acceleratia_initiala=5; sus=50; jos=440; coloana=320;
var inaltimea,pozitia,acceleratia: Integer;
begin
  OpenGraph; Line(0,jos+marimea+1,GetMaxX,jos+marimea+1);
  SetWriteMode(XorPut); { 1 } inaltimea:=sus;
  repeat
    pozitia:=inaltimea; acceleratia:=acceleratia_initiala;
    while pozitia<=jos do
      begin
        SetColor(culoarea); Circle(coloana,pozitia,marimea);
        Delay(pauza); SetColor(0); Circle(coloana,pozitia,marimea);
        Inc(pozitia,acceleratia); Inc(acceleratia,variatia_acceleratiei)
      end;
    Sunet; SetColor(culoarea); Circle(coloana,jos,marimea);
    Delay(pauza); SetColor(0); Circle(coloana,jos,marimea);
    Inc(inaltimea,variatia_inaltimii)
  until inaltimea>=jos;
  SetWriteMode(0); SetColor(culoarea); Circle(coloana,jos,marimea);
  ReadKey; CloseGraph
end.
```

Programul `PingPong2` realizează acelaşi lucru, însă pentru o bilă reprezentată sub forma unui cerc haşurat. Se folosesc procedurile `GetImage`, care preia imaginea inițială a figurii, şi apoi `PutImage`, care afişează imaginea în toate pozițiile de pe "traseu". Figura este preluată cu o mică margine care va determina ştergerea imaginii din vechea poziție. Acest procedeu are şi el un dezavantaj, acela că nu se pot prelua imagini prea mari în memorie. De asemenea, trebuie avut grijă ca marginea aleasă să fie suficientă pentru ca bila să nu lase urme; dacă în cazul unei mişcări rectilinii nu apar probleme, nu acelaşi lucru se întâmplă

într-o cădere liberă, care este o mişcare rectilinie uniform variară. Astfel, de pildă, dacă se pune valoarea 2 pentru `variatia_acceleratiei`, atunci bila va lăsa unele urme, ca în desenul de mai jos. Nu acelaşi lucru se întâmplă şi în cazul valorii 1, aşa cum apare în program.

```
{$X+}
program PingPong2;
{ animatie cu PutImage }
uses Crt, Graph, Graph1;
procedure Sunet;
begin Sound(200); Delay(100); NoSound end;
const variatia_inaltimii=50; variatia_acceleratiei=2;
      culoarea=Yellow; acceleratia_initiala=5; sus=50; jos=440; coloana=320;
      marimea=30; pauza=20;
var inaltimea,pozitia,acceleratia: Integer; Imaginea: Pointer;
    MarimeaImaginii: Word;
begin
  OpenGraph; SetColor(culoarea); SetFillStyle(CloseDotFill,culoarea);
  Circle(2*marimea,2*marimea,marimea);
  FloodFill(2*marimea,2*marimea,culoarea);
  MarimeaImaginii:=ImageSize(0,0,4*marimea,4*marimea);
  GetMem(Imaginea,MarimeaImaginii);
  GetImage(0,0,4*marimea,4*marimea,Imaginea^);
  ClearView(0,0,4*marimea,4*marimea); SetWriteMode(XorPut); { 1 }
  inaltimea:=sus;
  repeat
    pozitia:=inaltimea; acceleratia:=acceleratia_initiala;
    while pozitia<=jos do
      begin
        PutImage(coloana-2*marimea,pozitia-2*marimea,Imaginea^,NormalPut);
        Delay(pauza); Inc(pozitia,acceleratia);
        PutImage(coloana-2*marimea,pozitia-2*marimea,
                 Imaginea^,NormalPut);
        Inc(acceleratia,variatia_acceleratiei)
      end;
    Sunet; Inc(inaltimea,variatia_inaltimii);
    ClearView(coloana-2*marimea,
              jos-2*marimea,coloana+2*marimea,jos+2*marimea);
  until inaltimea>=jos;
  FreeMem(Imaginea,MarimeaImaginii); SetWriteMode(0);
  ReadKey; CloseGraph
end.
```



Transformări de imagini

Programele pe care urmează să vi le prezintăm şi explicăm în continuare sunt încercări timide de *morphing*. Prin **morphing** se înţelege transformarea unei imagini date în altă imagine. Algoritmii de morphing sunt dintre cei mai sofisticaţi, mai ales în cazul în care imaginile sunt memorate sub forma unor hărţi de biţi, deoarece trebuie mai întâi să se realizeze nişte recunoaşteri de forme, lucru nu tocmai uşor.

În cazul nostru, nu vom avea de a face cu desene reprezentate sub forma de hărţi de biţi, ci de desene văzute ca o colecţie de suprafeţe de diferite culori. O suprafaţă este, de fapt o curbă, reprezentată de o linie poligonală, care are o anumită culoare şi care eventual poate fi umplută (haşurată) în respectiva culoare.

Să zicem că avem două astfel de desene, care au acelaşi număr de suprafeţe, suprafeţele corespunzătoare având acelaşi număr de puncte, eventual aceleaşi culori şi acelaşi stil de umplere (`da` sau `nu`). Transformarea primului desen în cel de-al doilea presupune transformarea fiecărei suprafeţe (linie poligonală sau curbă) a primului desen în cea corespunzătoare din cel de al doilea desen.
Acest lucru este realizat de procedura `TransfPic`, din unit-ul `uAnim`, pe care îl listăm în paginile ce urmează. El cuprinde şi alte proceduri de lucru cu desene (editare, modificare a unui desen, salvare, restaurare etc) şi este mult comentat, încât nu credem că veţi avea probleme în înţelegerea lui. El foloseşte unit-ul `uMouse` de lucru cu mouse-ul, despre care am vorbit deja. În plus, deoarece animaţia care se va realiza va folosi modul grafic `640×350×16` (`VGAMed` sau `EGA`), ce permite folosirea a două pagini video simultan, am rescris procedura `OpenGraph` de iniţializare grafică pentru acest mod.

Cele două pagini video sunt numerotate cu `0` şi `1`. Dacă sa alege ca pagină vizibilă (adică cea care este pe ecran) pagina p (p fiind `0` sau `1`), iar ca pagină activă (adică cea în care au loc în realitate scrierile grafice) pagina `1-p`, atunci vom putea scrie în pagina `1-p`, fără ca ochiul nostru să sesizeze ceva pe ecran. Schimbând cele două pagini, adică făcând-o pe cea activă şi vizibilă, vom vedea ce desenasem.

Alegerea paginii active (care inițial este `0`) se realizează cu **`SetActivePage(nr_pag)`**, iar alegerea paginii vizibile (care inițial este tot `0`) cu **`SetVisualPage(nr_pag)`**.

```
unit uAnim;
{ in cele ce urmeaza vom considera sinonime expresiile
  "suprafata", "curba" si "linie poligonala" }
interface
  type YesNo=(no,yes,soso); {stilul de desenare suprafata,  desen:
              no = doar "curba" (linia poligonala), in alb;
              soso = doar "curba", in culoarea ei;
              yes = si curba si continutul ei, in culoarea respectiva}
  type surface=record
{o suprafata este o linie polig, având un nr de puncte, fiecare punct fiind
memorat prin cele doua coordonate ale sale: pt o suprafata S, prin S[3,2]
vom intelege coordonata a II-a(deci y) a celui de-al 3-lea punct}
                 Pct:array[1..30,1..2] of Integer;
                 NrPct: Byte; { numarul de puncte }
                 Fil,Cul: Byte { Fil = 0 (neumplut) sau 1 (umplut }
                            { Cul = 0..15 = culoarea suprafetei }
               end;
               desen=record
{ un desen este o multime de NrSurf suprafete }
                     Surf: array[1..20] of surface;
{ pt un desen D, prin Surf[2] vom intelege a 2-a linie poligonala a sa }
                      NrSurf: Byte
                    end;
  procedure OpenGraph;
{ initializeaza modul grafic EGA, sau VGA mediu, cu 16 culori si doua pagini
video, care pot fi interschimbate, pentru a crea animatie }
  procedure CreatePic(var D: desen);
{creaza interactiv un desen D, folosind mouse-ul }
  procedure DrawPic(D: desen; stil: YesNo);
{ afiseaza un desen D, in stilul stil }
  procedure TransfPic(D1, D2: desen; n: Byte; stil: YesNo);
{ transforma, prin translatie de puncte, des D1 in des D2; transformarea
  este realizata in n pasi, iar desenarile se fac in stilul precizat }
  procedure ModifyPic(var D: desen);
{ modifica un desen D, in mod interactiv, prin modificarea, pe rând
  a tuturor suprafetelor  ce il compun }
  procedure SavePic(D: desen; fis: String);
{ salveaza un desen D intr-un fisier fis }
  procedure LoadPic(var D: desen; fis: String);
{ incarca dintr-un fisier fis un desen, in variabila D }
  var PauzaDeAnimatie: Integer; { necesara lui TransfPic }

implementation
  uses Graph, uMouse, Crt;
  var pag: Byte; { numarul paginii curente ce se vizualizeaza }
  procedure Beep; { avertizare sonora }
  begin Sound(200); Delay(100); NoSound end;
  function Apartine(x,y,x1,y1,x2,y2: Integer): Boolean;
  {verifica apartenenta unui punct (x,y) la un dreptunghi (x1,y1)-(x2,y2)}
  begin  Apartine := (x1<=x) and (x<=x2) and (y1<=y) and (y<=y2)  end;
  procedure OpenGraph;
  { initializare grafica speciala }
```

```
  var gd,gm: Integer;
  begin
    gd:=VGA; gm:=VGAmed; InitGraph(gd,gm,'C:\PP\BGI'); pag:=0;
  end;
  procedure Bara;
  { sterge o portiune din partea stinga sus a ecranului, loc unde se fac
    unele citiri in cadrul procedurilor de creare si modificare de desen }
  begin
       SetViewPort(0,0,150,60,ClipOn); ClearViewPort;
       SetViewPort(0,0,GetMaxX,GetMaxY,ClipOn)
  end;
  procedure DrawSurface(S: surface; stil: YesNo);
  { deseneaza o suprafata S, in stilul stil }
  var cc: Byte;
  begin
    cc:=GetColor; { culoarea dinaintea apelului procedurii }
    case stil of
         yes: begin
         {daca este stilul "yes", atunci se deseneaza si linia poligonala}
                SetColor(S.Cul); DrawPoly(S.NrPct, S.Pct);
         {dar se umple si poligonul}
                SetFillStyle(S.Fil, S.Cul); FillPoly(S.NrPct, S.Pct)
              end;
          no: begin {desenare simpla cu alb}
                SetColor(White); DrawPoly(S.NrPct, S.Pct)
              end;
        soso: begin {desenare simpla, in culoarea proprie}
                SetColor(S.Cul); DrawPoly(S.NrPct, S.Pct)
              end
    end;
    SetColor(cc) {restaurarea culorii dinaintea apelului procedurii}
  end;
  procedure CreateSurface(var S: surface);
{ aceasta procedura creeaza interactiv, cu ajutorul mouse-ului, o linie
  polig S, de cul S.Cul, care poate fi umpluta (S.Fil=1) sau nu (S.Fil=0) }
  var b,x1,y1,x2,y2,xs,ys: Integer;
  begin
    S.NrPct:=1; { se asteapta fixarea primului punct al lui S }
    repeat MouseData(b,xs,ys) until b=1;
    x1:=xs; y1:=ys; S.Pct[S.NrPct,1]:=xs; S.Pct[S.NrPct,2]:=ys;
    { se memoreaza coordonatele acestui prim punct }
    repeat
      SetWriteMode(1); { se alege modul de scriere XorPut }
      MouseHide;
      repeat
{se uneste penultimul pct (x1,y1) cu cel actual, care se determina cu mouse-
ul, prin apasarea unui buton; observati cum se deseneaza liniile
intermediare, prin doua apeluri ale lui Line, in modul de scriere XorPut}
        MouseData(b,x2,y2);Line(x1,y1,x2,y2);MouseShow;Delay(50);MouseHide;
        {se afiseaza putin mouse-ul, pentru a vedea unde ne aflam}
        Line(x1,y1,x2,y2);
      until b in [1,2];
      SetWriteMode(0); { se revine la modul normal de scriere grafica }
      if b=1 then
         begin {apasarea butonului din stânga inseamna un nou punct}
           Delay(300); Inc(S.NrPct);
           S.Pct[S.NrPct,1]:=x2; S.Pct[S.NrPct,2]:=y2;
   {se memor coord acestui nou pct, apoi se des noua componenta a lin polig}
           Line(x1,y1,x2,y2); MouseShow; x1:=x2; y1:=y2
         end
    until b=2; {apas butonului din dreapta inceteaza desenarea suprafetei }
    Beep; {un zgomot de avertizare ca desenarea s-a sfirsit}
```

```
    MouseHide; Bara;
    { in continuare se introduc date referitoare la tipul suprafetei }
    OutTextXY(10,0,'Solid=1/Empty=0 ? '); { este umpluta sau nu }
    GoToXY(3,2); ReadLn(S.Fil);
    OutTextXY(10,27,'Culoare ? '); { ce culoare are }
    GoToXY(3,4);ReadLn(S.Cul); MouseShow;
    DrawSurface(S,yes); { o desenam, in stilul "yes" }
    SetColor(White); MouseShow
  end;
  procedure ModifySurface(var S: surface);
  { aceasta procedura permite modificarea unei suprafete S, utilizind mouse-
    ul; modif se face prin "agatarea" diferitelor pcte ale supraf si
    modificarea poz lor; acest fapt va duce si la modif liniilor in care
    intra acest punct, ca extremitate }
  const eps=3; {precizia in pixeli, pt agatarea cu mouse-ul a punctelor}
  var b,x,y,i,x_1,y_1,x_2,y_2: Integer;
{ b=but mouse; (x,y)=coordonate mouse (in preajma /) punctului agatat,
  (x_1,y_1) si (x_2,y_2) sint punctele vecine celui agatat }
     gasit: Boolean; { s-a agatat un punct sau nu ? }
  begin
    SetColor(White);
    repeat
      MouseData(b,x,y); i:=1; gasit:=False;
      { se cauta daca s-a agatat un punct sau nu }
      while (i<=S.NrPct) and (not gasit) do
       if (b=1) and Apartine(x,y,S.Pct[i,1]-eps,S.Pct[i,2]-eps,
                             S.Pct[i,1]+eps,S.Pct[i,2]+eps) then
          begin
            gasit:=True;  { da s-a agatat punctul al i-lea,
            deci coordonatele sale sunt: S.Pct[i,1] si S.Pct[i,2] }
            Beep; { se semnalizeaza sonor acest lucru }
            SetWriteMode(1); MouseHide; Delay(300);
            {se calculeaza coordonatele vecinilor lui}
            if i<S.NrPct then begin x_2:=S.Pct[i+1,1];y_2:=S.Pct[i+1,2] end
            else begin x_2:=-1; y_2:=-1 end;
            if i>1 then begin x_1:=S.Pct[i-1,1]; y_1:=S.Pct[i-1,2] end
            else begin x_1:=-1; y_1:=-1 end;
            { se traseaza liniile intre punctul gasit si vecinii sai}
            if x_1+y_1<>-2 then Line(x_1,y_1,S.Pct[i,1],S.Pct[i,2]);
            if x_2+y_2<>-2 then Line(S.Pct[i,1],S.Pct[i,2],x_2,y_2);
            if (x_1+y_1<>-2) or (x_2+y_2<>-2) then
               Line(S.Pct[i,1],S.Pct[i,2], S.Pct[i,1],S.Pct[i,2]);
            repeat
              { se stabilesc alte coordonate (x,y) pentru punctul i }
              MouseData(b,x,y); S.Pct[i,1]:=x; S.Pct[i,2]:=y;
              { tot asa, se deseneaza liniile intre punctul i, care acum
                va avea coordonatele  (x,y) ale mouse-ului, si vecinii sai
                (x_1,y_1), respectiv (x_2,y_2), daca acestia exista
                (lucru controlat prin suma coordonatelor <> -2 }
              if x_1+y_1<>-2 then Line(x_1,y_1,x,y);
              if x_2+y_2<>-2 then Line(x,y,x_2,y_2);
              MouseShow; Delay(50); MouseHide;
              if x_1+y_1<>-2 then Line(x_1,y_1,x,y);
              if x_2+y_2<>-2 then Line(x,y,x_2,y_2)
            until b=1;
            SetWriteMode(0);
            if x_1+y_1<>-2 then Line(x_1,y_1,x,y);
            if x_2+y_2<>-2 then Line(x,y,x_2,y_2);
            S.Pct[i,1]:=x; S.Pct[i,2]:=y;
            { in sfirsit ne-am ales noile coordonate }
            MouseShow; Delay(500); b:=0
          end
```

```
    else i:=i+1 { daca nu e punctul i cel agatat, poate e urmatorul }
  until b=2; { butonul 2, cel din dreapta, termina procedura }
  Beep;Bara; {...nu inainte de a se citi stilul hasurii si a culorii}
  OutTextXY(10,0,'Solid=1/Empty=0 ? '); GoToXY(3,2); ReadLn(S.Fil);
  OutTextXY(10,27,'Culoare ? '); GoToXY(3,4);ReadLn(S.Cul)
end;

procedure CreatePic(var D: desen);
{ creeaza un desen D, creând pe rând, toate suprafetele
  (liniile poligonale/curbele) ce il compun}
var i: Integer;
begin
  MouseHide; Bara; {la inceput se precizeaza cite suprafete are desenul}
  OutTextXY(0,0,'Cite suprafete ?'); GoToXY(1,2); ReadLn(D.NrSurf);
  ClearDevice; MouseShow; for i:=1 to D.NrSurf do CreateSurface(D.Surf[i])
end;
procedure DrawPic(D: desen; stil: YesNo);
{ afis un des D, desenând toate supraf ce il compun, in acelasi stil }
var i: Byte;
begin for i:=1 to D.NrSurf do DrawSurface(D.Surf[i], stil) end;
procedure TransfPic(D1, D2: desen; n: Byte; stil: YesNo);
{ transforma un desen D1 intr-un altul D2, in n pasi;
   numarul de suprafete ale lui D2 este acelasi ca si a lui D1;
   numarul de puncte ale fiecarei suprafete a lui D2, ca si
   parametrii Fil si Cul sunt aceleasi ca si la D1 }
var S1,S2,S: surface; k,i,j: Integer;
begin
  MouseHide;
  for i:=1 to n do { au loc n transformari, bazate pe translatii }
    begin
      for k:=1 to D1.NrSurf do
       { se transforma, la pasul i, toate cele D1.NrSurf suprafete,
         dupa acelasi model ca si in procedura TransfSurface }
       begin
         S1:=D1.Surf[k]; S2:=D2.Surf[k]; S.NrPct:=S1.NrPct;
         S.Fil:=S2.Fil; S.Cul:=S2.Cul;
         for j:=1 to S1.NrPct do
          begin
            S.Pct[j,1]:=Round(S1.Pct[j,1]+i*(S2.Pct[j,1]-S1.Pct[j,1])/n);
            S.Pct[j,2]:=Round(S1.Pct[j,2]+i*(S2.Pct[j,2]-S1.Pct[j,2])/n)
          end;
         DrawSurface(S,stil);
       end;
      { aici se realizeaza stergerea paginii grafice care nu se vede }
      SetVisualPage(1-pag); SetActivePage(pag); ClearDevice;
      Delay(PauzaDeAnimatie); { o mica pauza }
      pag:=1-pag { inversare de pagini grafice }
    end;
  MouseShow
end;
procedure ModifyPic(var D: desen);
{ modificarea ununi desen D inseamna modificarea tuturor celor
   D.NrSurf linii poligonale ce il compun }
var i: Byte;
begin for i:=1 to D.NrSurf do ModifySurface(D.Surf[i]) end;
procedure SavePic(D: desen; fis: String);
{ salvarea datelor unui desen D intr-un fisier fis; este procesul invers
  lui LoadPic, pe care il prezentam mai jos }
var i,j: Byte; F: Text;
begin
  Assign(F,fis); Rewrite(F); WriteLn(F,D.NrSurf);
  for i:=1 to D.NrSurf do
```

```
      with D.Surf[i] do
        begin
          WriteLn(F,Fil); WriteLn(F,Cul); WriteLn(F,NrPct);
          for j:=1 to NrPct do
              begin WriteLn(F,Pct[j,1]); WriteLn(F,Pct[j,2]) end
        end;
    Close(F)
  end;
  procedure LoadPic(var D: desen; fis: String);
  { incarcarea unui desen D dintr-un fisier fis presupun citirea din fisier
    a numarului de suprafete, apoi, pentru fiecare suprafata in parte,
    se citesc atributele Fil si Cul, dupa care se citeste câte puncte sunt;
    in sfârsit se citesc coordonatele tuturor punctelor acelei suprafete }
  var i,j: Byte; F: Text;
  begin
    Assign(F,fis); Reset(F); { deschiderea fisierului }
    ReadLn(F,D.NrSurf);
    for i:=1 to D.NrSurf do
      with D.Surf[i] do
        begin
          ReadLn(F,Fil); ReadLn(F,Cul); ReadLn(F,NrPct);
          for j:=1 to NrPct do
              begin ReadLn(F,Pct[j,1]); ReadLn(F,Pct[j,2]) end
        end;
    Close(F)
  end;
begin PauzaDeAnimatie:=0 { valoarea implicita a pauzei } end.
```

Iată şi un program în care utilizatorul va edita un desen `D1`. Apoi `D1` este copiat în `D2`, pe care utilizatorul îl va modifica. În final se salvează ambele desene, iar apoi se încarcă şi se transformă din unul în celălalt.

```
program EditorDeAnimatie;
uses Graph, uMouse, Crt, uAnim;
var D1,D2: desen;
begin
  OpenGraph; MouseInit; CreatePic(D1); SavePic(D1,'D1.PIC');
  Sound(100); Delay(50); NoSound;
  MouseHide; ClearDevice; MouseShow;Delay(400); LoadPic(D2,'D1.PIC');
  MouseHide;DrawPic(D2,yes); MouseShow; ModifyPic(D2);SavePic(D2,'D2.PIC');
  ClearDevice; LoadPic(D1,'D1.PIC'); LoadPic(D2,'D2.PIC');
  repeat
    TransfPic(D1,D2,20,yes); Delay(1500); ClearDevice;
    TransfPic(D2,D1,20,yes); Delay(1500)
  until keypressed;
  CloseGraph
end.
```

Cel de al doilea program (`Morphing`[1]) pe care vi-l prezentăm în continuare, va trebui să fie apelat din linia de comandă cu doi parametri, reprezentând numele celor două imagini între care se va face morphing, de exemplu:

```
C:\>morphing d1 d2
```

(Trebuie să existe pe disc fişierele `D1.PIC` şi `D2.PIC`, create de dumneavoastră prin editare, cu primul program, de pildă.).

```
program Morphing;
{ se ruleaza cu doi parametri, reprezentând numele celor doua PIC-uri;
  de ex.: C:\>morphing omhom1 omhom2 sau: C:\>morphing inima1 inima2 }
uses uAnim, Crt, Graph;
```

[1] cu extensia PAS, iar după compilare cu **`TPC /m morphing`** se obține un executabil **`morphing.exe`**

```
var D1,D2: desen;
begin
  OpenGraph; LoadPic(D1,ParamStr(1)+'.PIC'); LoadPic(D2,ParamStr(2)+'.PIC');
  repeat
    TransfPic(D1,D2,20,yes); Delay(1500); ClearDevice;
    TransfPic(D2,D1,20,yes); Delay(1500)
  until KeyPressed; CloseGraph
end.
```

Dacă nu aveți destul talent astfel încât să realizați nişte desene interesante, atunci vă prezentăm mai jos conținutul a două fişiere, unul reprezentând un om, iar celălalt o casă, fişiere numite `OMHOM1.PIC`, respectiv `OMHOM2.PIC`. Ele sunt fişiere text, care conțin fiecare câte un număr pe fiecare linie, dar, din motive obiective, listăm aceste numere pe mai multe rânduri, despărțite prin virgule, rămânând în sarcina dumneavoastră să scrieți fişierele.

- pentru fişierul `OMHOM1.PIC`:

6, 1, 14, 18, 201, 81, 201, 81, 165, 104, 164, 134, 178, 169, 205, 204, 239, 225, 269, 230, 300, 231, 321,
224, 340, 205, 360, 165, 367, 137, 347, 105, 309, 75, 269, 67, 229, 70, 205, 80, 1, 1, 8, 213, 110, 213,
110, 202, 115, 209, 122, 225, 123, 239, 119, 231, 111, 215, 110, 1, 1, 8, 299, 109, 299, 109, 284, 114,
297, 122, 317, 122, 327, 114, 314, 108, 301, 108, 1, 6, 20, 163, 101, 163, 101, 121, 34, 161, 49, 168, 20,
201, 44, 227, 9, 250, 42, 277, 4, 298, 43, 341, 17, 328, 53, 370, 33, 349, 103, 311, 74, 272, 66, 229, 69,
199, 81, 165, 101, 165, 101, 1, 3, 6, 256, 137, 256, 137, 252, 178, 279, 178, 263, 136, 257, 136, 1, 4, 10,
235, 199, 235, 199, 243, 206, 259, 210, 281, 209, 292, 203, 297, 197, 269, 201, 253, 201, 237, 198

- pentru fişierul `OMHOM2.PIC`:

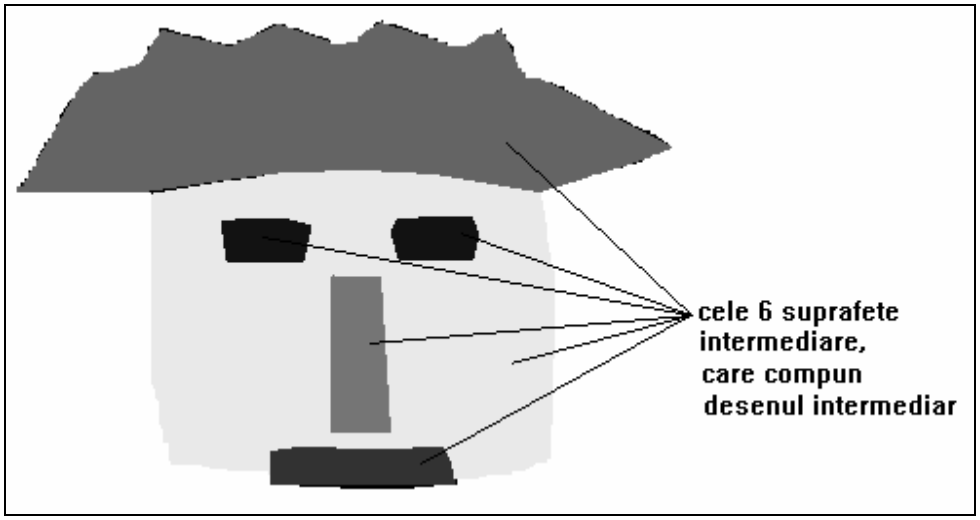
6, 1, 14, 18, 215, 91, 201, 92, 155, 92, 155, 132, 155, 201, 155, 241, 237, 241, 266, 241, 309, 241, 346,
241, 374, 241, 371, 165, 367, 137, 365, 91, 309, 91, 269, 91, 231, 90, 215, 91, 1, 1, 8, 217, 107, 207,
107, 195, 107, 196, 132, 243, 132, 243, 107, 228, 107, 217, 107, 1, 1, 8, 295, 107, 282, 108, 284, 114,
284, 131, 328, 131, 329, 115, 329, 107, 297, 107, 1, 6, 20, 151, 91, 63, 91, 130, 38, 151, 23, 165, 12,
201, 12, 226, 12, 259, 12, 278, 12, 319, 12, 340, 12, 358, 23, 454, 90, 365, 90, 304, 90, 268, 90,1 229, 90,
197, 91, 152, 91, 157, 89, 1, 3, 6, 253, 136, 252, 137, 254, 230, 284, 230, 284, 137, 255, 137, 1, 4, 10,
251, 232, 236, 232, 213, 232, 206, 253, 331, 253, 324, 242, 315, 232, 283, 232, 269, 232, 253, 232 .

După ce ați creat, cu mare grijă, fişierele cuprinzând numerele de mai înainte şi ați compilat (cu TPC) programul `Morphing`, încercați o rulare cu comanda:

```
C:\>morphing omhom1 omhom2
```

şi vedeți ce se întâmplă.

În figura următoare este prezentat un moment din timpul rulării acestui program, moment în care avem pe ecran un desen intermediar, care nu reprezintă nici un om, dar nici o casă. Programul se termină la apăsarea unei taste. Dacă execuția sa v-a amuzat, atunci poate vă interesează care sunt bazele teoretice ale acestui program.



În cele ce urmează vă vom prezenta o procedură care realizează o transformare a liniei poligonale S1 în linia poligonală S2, având acelaşi număr de puncte ca şi S1. Ideea care stă la baza acestei proceduri este foarte simplă.

Cum fiecare linie poligonală are un număr de puncte unite între ele prin segmente de dreaptă, ceea ce va trebui să facem pentru ca S1 să ajungă în postura lui S2 este să facem de aşa natură ca fiecare punct a lui S1 să ajungă în punctul corespunzător din S2. Acest lucru se va realiza în n pasi, prin translaţii; la un anumit pas, punctele de pe S1 se vor afla în poziţii "intermediare" (temporare). Fie P un punct dat al lui S1; la pasul i, lui P îi corespunde un punct intermediar P'. Dacă vecinii lui P în S1 sunt A şi B, iar acestora le corespund, la pasul i, punctele intermediare A', respectiv B', atunci va trebui să îl unim pe P' cu A' şi cu B'.

Dacă P are coordonatele (x1,y1) în S1, punctul P' va avea, la pasul i, coordonatele (x',y'), cu x' şi y' daţi de formulele:

x'=x1+i*(x2-x1) şi y'=y1+i*(y2-y1),

în care (x2,y2) reprezintă coordonatele punctului în care trebuie să ajungă punctul P în cei n pasi, adică coordonatele acestui punct în cadrul liniei poligonale S2.

Iată, deci, procedura:

```
procedure TransfSurface(S1, S2: surface; n: Byte);
var S: surface; { o linie poligonala a punctelor intermediare }
    i,j: Integer;
begin
  SetVisualPage(1-pag); MouseHide; SetVisualPage(pag); MouseHide;
  S.NrPct:=S1.NrPct;
{ linia polig. intermediara are acelasi numar de puncte ca si S1 si S2 }
  for i:=1 to n do
      begin
        for j:=1 to S1.NrPct do
            begin
              { se calculeaza coordonatele punctului intermediar }
              S.Pct[j,1]:=Round(S1.Pct[j,1]+i*(S2.Pct[j,1]-S1.Pct[j,1])/n);
              S.Pct[j,2]:=Round(S1.Pct[j,2]+i*(S2.Pct[j,2]-S1.Pct[j,2])/n)
            end;
        SetVisualPage(1-pag); SetActivePage(pag);
```

```
        ClearDevice;  { se sterge pagina grafica nevizibila }
        DrawSurface(S,yes);
      {se des suprafata intermediara, aici in stilul "yes", deci complet}
        Delay(PauzaDeAnimatie);
        pag:=1-pag { se inverseaza paginile grafice }
      end;
   SetVisualPage(1-pag); MouseShow; SetVisualPage(pag); MouseShow
end;
```

Procedura de mai sus stă la baza procedurii care transformă un desen în altul, prezentată în unit-ul uAnim.

Animație profesională. Formatul fişierelor FLI

În primele părți ale acestei secțiuni s-au prezentat trei tehnici simple de animație şi anume:

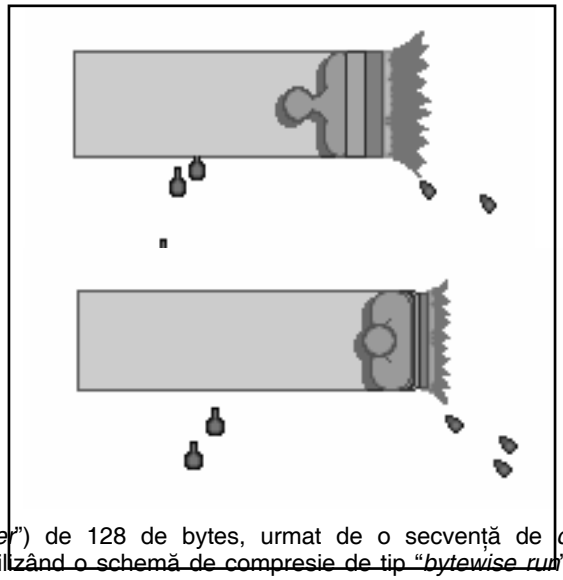
- redesenarea în vechea poziție cu modul de scriere XOR, care determină ştergerea desenului şi scrierea în noua poziție; (se foloseşte SetWriteMode(1));
- alternarea paginilor video; (se folosesc SetVisualPage şi SetActivePage) (în programul de morphing);
- prelucrarea directă a diferite zone dreptunghiulare de ecran (se folosesc GetImage şi PutImage).

Însă este evident pentru oricine că, de cele mai multe ori, aceste tehnici nu dau rezultate formidabile, nici chiar dacă se combină între ele. Ele prezintă, de fapt şi unele dezavantaje evidente: prima duce la clipirea imaginii pe ecran, a doua la o redesenare care încetineşte ritmul animației, iar a treia la o alocare de memorie dinamică destul de mare ca dimensiuni, în heap.

Programele profesionale de animație, cum este *Autodesk Animator*, prezintă un moment de animație prin suprapunerea a mai multor cadre (frame-uri), unul după altul. Acestea se citesc de pe disc, dintr-un fişier cu extensia FLI (sau, mai nou, FLC), care se numeşte "*flic*" şi care are memorat paleta de culori folosite, precum şi primul cadru din animație, apoi, pentru celelalte, diferențele față de cadrul precedent.

În cele ce urmează ne propunem să prezentăm structura fişierelor FLI folosite de *Autodesk Animator* şi *Autodesk 3D Studio* propunând cititorului, ca exercițiu să scrie un program de vizualizare a unor astfel de flic-uri, care să urmeze întocmai prezentarea teoretică a structurii fişierelor FLI.

Detaliile unui fişier FLI (un "flic") sunt relativ complexe, dar ideea de bază este simplă: nu se memorează părțile unui cadru (frame) care sunt aceleaşi ca şi în cadrul anterior. Nu pentru că acest lucru economiseşte spațiu, ci pentru că este mult mai rapid procesul de animație, efectuat prin suprapunerea unor cadre. Este mult mai rapid să laşi un pixel în pace, decît să-l setezi!
Iată, de exemplu, două cadre succesive (care nu diferă prea mult între ele) dintr-un fişier FLI:



Un fişier `FLI` are un *antet* ("*header*") de 128 de bytes, urmat de o secvență de *cadre*. Primul cadru este compresat utilizând o schemă de compresie de tip "*bytewise run*". Cadrele următoare sunt memorate prin diferența față de cadrul precedent. (Ocazional, primul cadru şi/sau celelalte sunt necompresate.). Există, de asemenea, la sfârşitul unui `FLI`, un cadru suplimentar ce conține diferența între ultimul cadru şi primul cadru.

Antetul unui fişier `FLI` este:

byte offset	mărime	nume	semnificatie
0	4	size	Lungimea fisierului, pentru programele care doresc să citească flic-ul cu totul, dacă este posibil
4	2	magic	Setat la valoarea AF11 (în hexa). Vă rugăm să folositi aici altă valoare, dacă schimbati formatul (chiar si pentru o altă rezolutie), astfel încât Autodesk Animator să nu se blocheze, încercând să citească flic-ul
6	2	frames	Numărul de cadre din FLI. Fisierele FLI au un continut de maxim 4000 cadre.
8	2	width	Lătimea ecranului (320).
10	2	height	Înăltimea ecranului (200).
12	2	depth	Adâncimea unui pixel (8).
14	2	flags	Trebuie să fie 0.
16	2	speed	Numărul de momente video între cadre..
18	4	next	Setat la 0.
22	4	frit	Setat la 0.
26	102	expand	Toate zero -- pentru dezvoltări viitoare.

Urmează cadrele (*frame*-urile), fiecare din ele având un antet propriu:

byte offset	mări-me	nume	semnificatie
0	4	size	Numărul de bytes în cadre. Pentru Autodesk Animator acesta este sub 64K.

4	2	magic	Intotdeauna hexazecimal F1FA.
6	2	chunks	Numărul de 'chunks' (“grupe”) în cadru.
8	8	expand	Spatiu pentru dezvoltări ulterioare. Toate zerouri.

După antetul cadrului vin grupele (“*chuncks*”) care formează cadrul, adică colecţii de informaţii ce specifică diferenţa faţă de cadrul anterior. Mai întâi vine o grupă de culoare, dacă paleta de culori s-a schimbat faţă de ultimul cadru. Apoi urmează o grupă de pixeli, dacă pixeliii s-au schimbat. Dacă cadrul este absolut identic cu precedentul, atunci nu va fi nici o grupă.

O grupă, la rândul ei, are un antet, urmat de date (culorile pixelilor). Antetul unei grupe este:

byte offset	mărime	nume	semnificatie
0	4	size	Numărul de bytes în această grupă.
4	2	type	Tipul de grupă (*chunk*) (vezi mai jos).

În mod curent, există cinci tipuri de grupe pe care le puteţi vedea într-un fişier `FLI`:

număr	nume	semnificatie
11	FLI_COLOR	Harta compresată a culorilor (paleta de culori).
12	FLI_LC	Linie compresată - cea mai întâlnită formă de compresie pentru orice cadru, mai putin pentru primul. Descrie diferentele de pixeli fată de cadrul precedent.
13	FLI_BLACK	Setează întregul ecran la culoarea 0 (apare doar la primul cadru).
15	FLI_BRUN	Compresie de tip “bytewise run-length” -- doar primul cadru.
16	FLI_COPY	Indică 64000 bytes necompresati care urmează. Pentru momentele în care compresia nu lucrează !

Schemele de compresie sunt toate orientate pe byte. Dacă datele de compresie dintr-un cadru se termină la o adresă impară, se mai inserează un pixel, astfel încât cadrele `FLI_COPY` vor începe întotdeauna la o adresă pară, ceea ce permite un acces direct în memoria video (DMA) mai rapid.

- **Grupa `FLI_COLOR`**

Primul word (cuvânt) (2 bytes) este numărul de *pachete* (“*packets*”) din această grupă. Acesta este urmat direct de pachete. Primul byte al unui pachet spune câte culori trebuie să fie sărite. Următorul byte ne spune câte culori se schimbă. Dacă acest byte este zero, el trebuie interpretat ca fiind 256. Apoi urmează 3 bytes pentru fiecare culoare în parte care se schimbă (unul pentru roşu, unul pentru verde şi unul pentru albastru).

- **Grupa `FLI_LC`**

Acesta este cel mai comun şi, de asemenea, cel mai complex gen de grupă (*chunck*). Primul word (cuvânt) (16 biţi) este numărul de linii, începând cu partea de sus a ecranului, care sunt identice cu cele din cadrul precedent. (De exemplu, dacă există modificări doar pe cea mai de jos linie a ecranului, vom avea aici valoarea `199`.). Următorul word este numărul de linii care se schimbă. Apoi urmează datele pentru schimbarea liniilor. Fiecare linie este compresată individual, printre altele aceasta determinând simplitatea vizualizării unui `FLI` la o mărime redusă.
Primul byte al unei linii compresate este numărul de pachete în respectiva linie. Dacă linia este neschimbată faţă de cea din ultimul cadru, acest număr este zero. Formatul unui pachet individual este:

```
skip_count
size_count
data
```

Aici, `skip_count` (contorul de salt) este un singur byte. Dacă mai mult de 255 pixeli trebuie să fie săriți, trebuie ca pachetul să fie descompus în două pachete. Şi `size_count` (contorul de mărime) este, de asemenea, un byte. Dacă acesta este pozitiv, aceasta înseamnă că mai mulți bytes de date (`data`) urmează şi vor fi copiate pe ecran, iar dacă este negativ, un singur byte urmează şi el este repetat de `-size_count` ori.
În cel mai rău caz, un cadru `FLI_LC` poate fi cam de 70K. Dacă el ajunge să fie de 60000 bytes sau chiar mai mult, *AutoDesk Animator* decide să renunțe la compresie (care este ineficientă) şi să salveze cadrul sub forma de `FLI_COPY`.

- **Grupa `FLI_BLACK`**

Acest tip este foarte simplu. Nu există date asociate. Această grupă (care "curăță" ecranul) este generată doar pentru primul cadru în *Autodesk Animator*, după ce a fost utilizată comanda `NEW` din meniul `FLIC`.

- **Grupa `FLI_BRUN`**

Aceste grupe sunt foarte asemănătoare celor de tip `FLI_LC`, dar nu apar salturile. Ele încep imediat cu date pentru prima linie a ecranului şi continuă, linie cu linie, în jos. Primul byte conține numărul de pachete în linia respectivă. Formatul pentru un pachet este:

```
size_count
data
```

Dacă `size_count` este pozitiv, `data` constă dintr-un singur byte care se repetă de `size_count` de ori. Dacă `size_count` este negativ, atunci există `-size_count` bytes de date, care sunt copiate pe ecran. În *Autodesk Animator*, dacă datele "compresate" semnalizează depăşirea a 60000 bytes, atunci cadrul este stocat sub forma unui `FLI_COPY`.

- **Grupa `FLI_COPY`**

Aceste grupe sunt reprezentate de 64000 bytes de date pentru citiri şi scrieri directe pe ecran.

Lecția 12
PROGRAMAREA ORIENTATĂ PE OBIECTE

1. Introducere

Programarea orientată pe obiecte (POO, OOP) este o tehnică de programare, un stil, ghidat de principii foarte riguroase, aşa cum este şi programarea structurată, însă aceste două tehnici nu se exclud una pe alta, ci, dimpotrivă, ele se completează reciproc în procesul realizării unor produse soft cât mai performante, cât mai complexe şi cât mai flexibile.

Pentru a înțelege noțiunea de **obiect** din informatică, trebuie să facem analogia cu obiectele concrete din din realitatea înconjurătoare. Să ne gândim, de pildă, la vehicule. Ele sunt obiecte bine cunoscute nouă, dar *clasa vehiculelor* este prea largă, aşa încât putem să o împărțim în cea a *vehiculelor aeriene*, cea a *vehiculelor terestre*, cea a *vehiculelor de apă* (marine şi... submarine) şi a *vehiculelor subpământene*.

La rândul ei, clasa vehiculelor aeriene s-ar putea împărți în *clasa avioanelor* şi *clasa elicopterelor*. Vehiculele terestre pot fi *de şosea* sau *pe cale ferată*. Cele pe cale ferată pot fi *în localități* (tramvaie) sau *între localități* (trenuri). Până acum toată clasificarea noastră a vehiculelor în clase şi subclase, poate fi reprezentată arborescent - sub forma din figura dată mai jos.

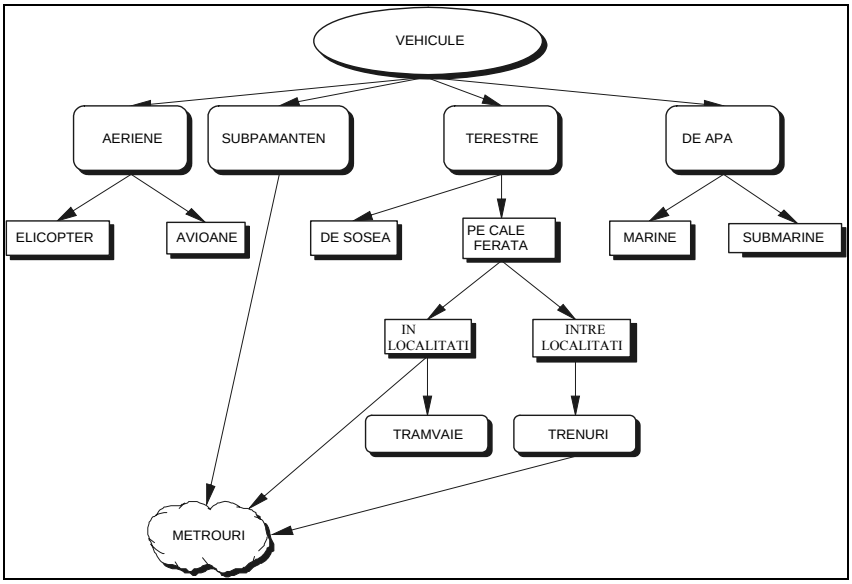
Am putea spune, astfel că avem clasa vehiculelor aeriene **derivată** din cea a vehiculelor (în general), iar din această clasă derivată se derivează clasa elicopterelor şi cea a avioanelor. La fel cu trenurile derivate din "*vehicule din localități*", clasa acestor vehicule fiind derivată din cea "*a vehiculelor pe cale ferată*", clasă derivată - la rândul ei - din cea "*a vehiculelor terestre*", al cărui **strămoş** este clasa generală "*vehicule*".

Putem spune că vehiculele pe cale ferată **moştenesc** toate caracteristicile vehiculelor (în general) şi, eventual, au caracteristici noi. Dar şi *metroul* este un tren, cu toate că merge pe sub pământ. Aşadar, dacă am considera şi clasa metrourilor, aceasta ar avea trăsăturile următoarelor clase: a trenurilor, a vehiculelor din localități şi cea a vehiculelor subpământene. Deci ar fi o **derivare multiplă** din mai multe clase.

În afară de această clasificare care se poate face chiar şi mai bine, pentru a evita eventualele derivări multiple, toate vehiculele (deci de la orice nivel) se caracterizează prin **structura** şi prin **funcționalitatea** lor.

Astfel, dacă la nivelul clasei generale a vehiculelor nu putem spune - referindu-ne la structură - decât că un vehicul are față, spate, număr de locuri şi *conducător*, când ne situăm la nivelul avioanelor sau al trenurilor putem deja vorbi despre *pilot* sau despre *mecanic de locomotivă*.

De asemenea, ne punem problema: ce funcții poate îndeplini un vehicul, adică ce poate să facă un vehicul sau ce se poate face cu un vehicul. Firește, orice vehicul, indiferent de clasa particulară în care ar putea fi încadrat, *se deplasează*, deci realizează acest lucru, chiar dacă la nivelul clasei generale "*vehicule*" nu putem defini prea detaliat sau prea clar ce înseamnă acest lucru, adică *cum* se realizează această deplasare, poate doar prin faptul că, îşi schimbă poziția în spațiu, dintr-un loc în altul. Dar dacă ne situăm în cadrul clasei elicopterelor, de pildă, ne dăm seama că deplasarea se face *prin zbor*, folosindu-se o *elice*, acționată de un motor. Astfel, devine clar modul de realizare a deplasării elicopterului, ba chiar apare o caracteristică structurală în plus, specifică elicopterelor şi nu tuturor vehiculelor, *elicea*.



Însă, pentru a vorbi despre un vehicul, trebuie ca acesta să existe, deci trebuie să fie creat sau *construit*. Când nu ne mai trebuie, îl *distrugem*.

Toată realitatea înconjurătoare este guvernată de astfel de "reguli", iar informaticienii au încercat - şi în mare parte au reuşit - să folosească toate acestea în programare. Astfel a apărut ideea de **obiect** în programare.

Ca şi în lumea reală, în informatică, un obiect ar fi caracterizat de structura sa - definită de anumite **câmpuri** - ca la tipul înregistrare, precum şi de funcționalitatea sa, definită de anumite **proceduri şi funcții** proprii clasei respective de obiecte.
Un obiect poate fi construit - dacă se foloseşte un **constructor**, precum şi distrus - cu un **destructor**.

Astfel, în Turbo-Pascal, apare - ca o extindere naturală a tipului `record` - noul tip **`object`**. Dacă vrem să definim o clasă a obiectelor de tip `tren`**,** va trebui să folosim o declarație de genul:

```
type
  Tren=object
         mecanic: String; {numele mecanicului de locomotivă}
         tip: (de_marfa,de_persoane); {cele trei câmpuri sunt ca la record}
         NV: Integer; {numarul de vagoane ale trenului}
         procedure Deplaseaza; {aceasta e o procedura ce poate fi executata}
       end;
```

Nu e mare filosofie! Ei bine, un tren este caracterizat de cele trei câmpuri ce-i determină structura: *numele mecanicului*, *tipul trenului* (aici ales dintr-o enumerare), precum şi de faptul că *se poate deplasa*, deci de acea procedură `Deplaseaza`. Singurele restricții sunt acelea că procedurile şi funcțiile (pot fi mai multe !) trebuie să figureze după toate câmpurile de structură, iar după ultima procedură trebuie scris simbolul "`;`" neapărat.

Acum dacă `T` este o variabilă de tip `Tren`, iar `MultimeDeTrenuri` un vector de trenuri: `var T: Tren; MultimeDeTrenuri: array[1..10] of Tren;` putem să ne referim la numele mecanicului din trenul `T` prin `T.mecanic`, precum şi la al treilea tren din `MultimeDeTrenuri`, prin `MultimeDeTrenuri[3]`, tipul acestui din urmă tren nefiind referit decât prin `MultimeDeTrenuri[3].tip`.

Pentru a spune că vrem să se execute procedura `Deplaseaza` asupra trenului `T` vom scrie pur şi simplu `T.Deplaseaza` şi la fel şi pentru al treilea tren din vector: `MultimeDeTrenuri[3].Deplaseaza`.
Cum s-ar descrie această procedură? Cum ea este ceva ce caracterizează nu numai pe `T` sau pe fiecare componentă a vectorului `MultimeDeTrenuri`, ci întreaga clasă a obiectelor de tip `Tren`, se va descrie în felul următor:

```
procedure Tren.Deplaseaza;
begin
        { ........ }
end;
```

În afară de astfel de proceduri, putem avea şi constructori şi destructori. Deşi pot fi consideraţi şi folosiţi pentru a iniţializa câmpurile de structură (**atributele**) unui obiect, ei se folosesc şi în alte scopuri

2. Definirea tipurilor obiect

Un tip obiect se declară ca un tip înregistrare, specificându-se toate atributele (câmpurile structurale) şi metodele (câmpurile funcționale) obiectelor. Ambele tipuri de câmpuri pot fi fie *publice* (implicit), fie *private*. Câmpurile private, fie că sunt atribute, fie că sunt metode, nu vor putea fi recunoscute în exteriorul obiectelor. Aşadar, dacă se declară un câmp atribut privat, asupra lui nu se va putea acționa decât din cadrul unei metode a obiectui respectiv. De asemenea, o metodă privată va putea fi apelată doar din cadrul altei metode a obiectului, nu şi din exterior. Metodele publice vor fi cele ce vor permite comunicarea între obiect şi mediul exterior, ele putând fi apelate şi din blocul în care a fost definit tipul obiect respectiv. La fel în cazul atributelor private.

Declararea unor câmpuri publice, respectiv private se face cu directivele **`public`**, respectiv **`private`**. Dacă nu se specifică, câmpurile se consideră implicit publice.

În cazul cel mai general, o clasă de obiecte se declară astfel:

```
type id_tip_obiect = object
                       public
                         câmpuri_publice;
                       private
                         câmpuri_private;
                     end;
```

O variabilă de acest tip se va declara: `var v:id_tip_obiect`.
Completa definire a tipului `id_tip_obiect` presupune şi definirea explicită a tuturor metodelor declarate.
De pildă, dacă vrem să definim un obiect grafic oarecare, pe care să putem să-l încadrăm într-o zonă dreptunghiulară determinată de `(x1,y1,x2,y2)`, iar obiectul să poată fi iniţializat, desenat, şters şi mutat cu mouse-ul, am putea scrie:

```
type TObiect = object
                 private
                   x1,y1,x2,y2: Integer;
                   procedure Display;
```

```
    procedure Clear;
  public
    procedure Init(x10,y10,x20,y20: Integer);
    procedure Move;
end;
```

În exterior nu ne interesează atributele obiectelor, nici procedurile de afişare, ştergere. Avem nevoie doar de procedura de inițializare, în care `x10`, `y10`, `x20` şi `x20` sunt coordonatele ce definesc poziția inițială a obiectului, precum şi de procedura de mutare a unui obiect cu mouse-ul.

Metoda `Init` va prelua valorile din lista de parametri şi le va da câmpurilor `x1`, `y1`, `x2` şi `y2`, apoi va afişa obiectul grafic respectiv, apelând la metoda privată `Display`, la care `Init` are acces.

```
procedure TObiect.Init;  { OI }
begin x1:=x10; y1:=y10; x2:=x20; y2:=y20; Display end;
```

Mutarea obiectelor va fi realizată cu mouse-ul într-o manieră asemănătoare celei de desenare a unui dreptunghi pe care o foloseam la programul `Desenator` din lecția precedentă. Ea va apela la metodele `Clear` (care va şterge obiectul din vechea poziție) şi `Display`, care îl va reafişa în noua poziție. Deocamdată nu ne interesează cum s-ar implementa `Display`, deoarece această metodă diferă de la obiect la obiect, cert este că `Display` va afişa ceva în zona dreptunghiulară `(x1,y1,x2,y2)`, ceva ce `Clear` va putea şterge cu `ClearViewPort`, procedură pe care am descris-o şi pus-o deja în biblioteca noastră grafică `Graph1`.

```
procedure TObiect.Clear; { OC }
begin ClearView(x1,y1,x2,y2) end;
procedure TObiect.Display;
begin { va fi definita ulterior } end;
procedure TObiect.Move;  { OM }
var b,px,py,x,y: Integer;
begin
  MouseData(b,px,py);
  if (b=1) and Apartine(px,py,x1,y1,x2,y2) then
     begin
       SetWriteMode(1); MouseHide;
       repeat
         MouseData(b,x,y); Rectangle(x1+x-px,y1+y-py,x2+x-px,y2+y-py);
         MouseShow; Delay(50); MouseHide;
         Rectangle(x1+x-px,y1+y-py,x2+x-px,y2+y-py);
       until b=0;
       SetWriteMode(0); Clear;
       x1:=x1+(x-px); y1:=y1+(y-py); x2:=x2+(x-px); y2:=y2+(y-py);
       Display; MouseShow
    end
end;
```

Să considerăm că obiectele grafice pe care dorim să le afişăm şi mutăm sunt nişte elipse. Atunci vom defini metoda `Display` a tipului `TObiect` în felul de mai jos:

```
procedure TObiect.Display; { OD }
begin Ellipse((x1+x2)div 2,(y1+y2)div 2,0,360,(x2-x1)div 2,(y2-y1)div 2)
end;
```

În acest moment, putem realiza deja un simplu program grafic, care să ne permită mutarea cu mouse-ul a unei elipse desenate pe ecran. Ea se "agață" cu butonul din stânga al mouse-ului din interiorul dreptunghiului ce ar încadra-o. Pe parcursul deplasării mouse-ului (cu butonul din stânga ținut apăsat), un dreptunghi va urma direcția de deplasare a mouse-ului. Când ne oprim (eliberăm butonul de mouse), dreptunghiul dispare, coordonatele acestuia se

actualizează, elipsa dispare din vechea poziție şi reapare în cea nouă. Programul este următorul, în care am pus în loc de proceduri, marcajele lor anterioare.

```
program ObiectGrafic;
uses Graph, Graph1, uMouse, Crt;
function Apartine(x,y,x1,y1,x2,y2: Integer): Boolean;
begin
  Apartine := (x1<=x) and (x<=x2) and (y1<=y) and (y<=y2)
end;
type TObiect=object
               private
                 x1,y1,x2,y2: Integer;
                 procedure Display;
                 procedure Clear;
               public
                 procedure Init(x10,y10,x20,y20: Integer);
                 procedure Move;
             end;
OI; OC; OD; OM;
var Obiect: TObiect;
begin
  OpenGraph; Obiect.Init(100,100,150,180);
  MouseInit; MouseShow;
  repeat Obiect.Move until KeyPressed;
  CloseGraph
end.
```

3. Derivare

În continuare, să considerăm că dorim să extindem obiectul grafic din exemplul anterior, punându-i un nou atribut şi anume raza unui cerculeț care să se deseneze în interiorul elipsei. Vom defini astfel, un nou obiect, care va avea toate caracteristicile celui anterior, plus atributul `raza`. Dar un astfel de obiect va trebui să aibă o nouă procedură de afişare, pe care să o numim, deocamdată, `DisplayC`, care va desena o elipsă şi un cerculeț de rază `raza`. Această procedură va desena un mic cerculeț, după care va apela procedura `Display`. Vom spune că elipsa cu cerculeț **moşteneşte** elipsa de tip `TObiect`. preluânindu-i toate câmpurile (inclusiv pe `Display`), şi are noi câmpuri: `raza` (atribut) şi `DisplayC` (metodă). Operația prin care clasa elipselor cu cerculeț moşteneşte clasa `TObiect`, a elipselor simple, se numeşte şi **derivare**. Se spune că tipul `TElipsaCerc` este **derivat** din tipul `TObiect`. Vom scrie, în general, acest lucru în felul următor:

```
type ob_derivat = object(ob_de_baza)
                    ........
                  end;
```

În cazul nostru concret avem:

```
program ObiecteGraficeDerivate;
....... { ca in programul precedent }
{ DERIVARE }
type TElipsaCerc = object(TObiect)
                     raza: Integer; { publice }
                     procedure DisplayC;
                   end;
{ METODA NOUA }
procedure TElipsaCerc.DisplayC;
begin
     Circle((x1+x2) div 2, (y1+y2) div 2, raza); Display
```

```
end;
var E: TElipsaCerc;
begin
     OpenGraph; E.Init(100,100,150,180);
     E.raza:=5; E.DisplayC; ReadLn; CloseGraph
end.
```

Fireşte, vom putea să folosim în continuare metoda `Move` a clasei părinte, însă nu trebuie ca, după mutare, să ne aşteptăm ca elipsa să cuprindă şi cerculeţul, deoarece metoda `Move` va apela metoda `Display` şi nu metoda `DisplayC` !

4. Polimorfism

Pentru a rezolva problema ivită în paragraful precedent, recurgem la **polimorfism**. Aceasta înseamnă că putem să avem în clasa derivată `TElipsaCerc` o metodă nouă `Display`, care să aibă acelaşi nume cu metoda din clasa părinte (`TObiect`), însă să execute alt lucru. Astfel, vom înlocui pe `TElipsaCerc.DisplayC` cu `TElipsaCerc.Display`. Însă, de această dată, va trebui să precizăm explicit în cadrul acestei metode că se apelează metoda cu acelaşi nume a clasei moştenite. De aceea vom scrie : `TObiect.Init`. Programul este dat mai jos. Vom obţine acelaşi rezultat ca şi în cel de mai înainte (secţiunea 3), însă cele două metode (din clasa părinte şi cea derivată) au acelaşi nume. În plus, am redenumit metoda `Init` care va permite, de această dată să punem câmpul `raza` ca privat (lucru mai indicat în programarea pe obiecte), deoarece `raza` va fi iniţializată de această metodă, rămasă publică.

După ce se desenează elipsa cu cerculeţ, se aşteaptă mutarea sa cu mouse-ul. Aceasta se realizează, însă cerculeţul nu se mută, din păcate, tocmai pentru că se apelează metoda `Init` a clasei părinte şi nu cea a clasei derivate...

```
program MetodePolimorfe;
 . . . . . .   { ca mai inainte }
type TElipsaCerc=object(TObiect)
                    private
                      raza: Integer;
                    public
                      procedure Init(x10,y10,x20,y20,raza0: Integer);
                      procedure Display;
                end;
procedure TElipsaCerc.Init;
begin
  TObiect.Init(x10,y10,x20,y20); { apel metoda clasei de baza }
  raza:=raza0 { initializarea razei cercului }
end;
procedure TElipsaCerc.Display;
begin
  TObiect.Display; Circle((x1+x2) div 2, (y1+y2) div 2, raza)
end;
var E: TElipsaCerc;
begin
  OpenGraph; E.Init(100,100,150,180,5); E.Display;
  MouseInit; MouseShow;
  repeat  E.Move  until KeyPressed; CloseGraph
end.
```

5. Metode virtuale

Pentru a rezolva şi ultima problemă pusă în paragraful anterior, se declară metoda `Display` (care este apelată de metoda de mutare a obiectelor `Move`) ca fiind **metodă virtuală** sau **dinamică**, folosind directiva **`virtual`**. În acest fel, metoda `Move`, când face apel la metoda `Display` va identifica în momentul execuţiei tipul obiectului pentru care trebuie să se facă afişarea şi, în funcţie de acesta, se va apela procedura `Display` corespunzătoare.

În acest caz, însă, orice redefinire a metodei, în cadrul claselor derivate, se va face folosind tot cuvântul `virtual`. De aesemenea, aceste clase vor trebui să aibe câte o metodă specifică, numită **`consructor`**, care creează în memorie o *tabelă de metode virtuale*.

Constructorul este prima metodă ce se apelează în cadrul obiectului şi are un rol dublu:

- după apelul constructorului, relativ la un obiect, o metodă care apare redefinită este apelată din cadrul obiectului pentru care s-a făcut apelul constructorului şi nu din cadrul obiectului din care s-a făcut apelul;
- în cadrul constructorului se pot face opţional şi alte operaţii, de pildă iniţializări, motiv pentru care în exemplul nostru vom considera drept constructor metoda `Init`.

Menţionăm că metodele de tip `constructor` nu pot fi virtuale. De asemenea, nu pot fi redefinite atributele (câmpurile structurale) dintr-o clasă de obiecte.

```
program MetodeVirtuale;
uses Graph, Graph1, uMouse, Crt;
function Apartine(x,y,x1,y1,x2,y2: Integer): Boolean;
begin Apartine := (x1<=x) and (x<=x2) and (y1<=y) and (y<=y2) end;
type TObiect = object
                 private
                   x1,y1,x2,y2: Integer;
                   procedure Display; virtual;
                   procedure Clear;
                 public
                   constructor Init(x10,y10,x20,y20: Integer);
                   procedure Move;
            end;
type TElipsaCerc=object(TObiect)
                   private
                     raza: Integer;
                   public
                     constructor Init(x10,y10,x20,y20,raza0:Integer);
                     procedure Display; virtual;
                 end;
constructor TElipsaCerc.Init;
begin
     TObiect.Init(x10,y10,x20,y20); raza:=raza0
end;
procedure TElipsaCerc.Display;
begin
     TObiect.Display; Circle((x1+x2) div 2, (y1+y2) div 2, raza)
end;
constructor TObiect.Init;
begin
     x1:=x10; y1:=y10; x2:=x20; y2:=y20; Display
end;
procedure TObiect.Clear;
begin ClearView(x1,y1,x2,y2) end;
procedure TObiect.Display;
begin
  Ellipse((x1+x2)div 2,(y1+y2)div 2,0,360,(x2-x1)div 2,(y2-y1)div 2)
```

```
end;
procedure TObiect.Move;
var b,px,py,x,y: Integer;
begin
  MouseData(b,px,py);
  if (b=1) and Apartine(px,py,x1,y1,x2,y2) then
     begin
       SetWriteMode(1); MouseHide;
       repeat
         MouseData(b,x,y); Rectangle(x1+x-px,y1+y-py,x2+x-px,y2+y-py);
         MouseShow; Delay(50); MouseHide;
         Rectangle(x1+x-px,y1+y-py,x2+x-px,y2+y-py);
       until b=0;
       SetWriteMode(0); Clear;
       x1:=x1+(x-px); y1:=y1+(y-py); x2:=x2+(x-px); y2:=y2+(y-py);
       Display; MouseShow
     end
end;
var O: TObiect; E: TElipsaCerc;
begin
  OpenGraph; E.Init(100,100,150,180,5); O.Init(200,200,270,240);
  E.Display; O.Display; MouseInit; MouseShow;
  repeat E.Move; O.Move until KeyPressed; CloseGraph
end.
```

Un exemplu mai complex care foloseşte metodele virtuale este dat de programul următor. Acesta permite mutarea cu mouse-ului a mai multor obiecte diferite, memorate în nişte vectori. Aceste obiecte sunt de tipurile `TDrept` şi respectiv `TDisc`, ambele clase fiind derivate din clasa generală `TObiect`, a cărei metodă `Display` este vidă, ea fiind redefinită la nivelul claselor ce o moştenesc.

Çi constructorii sunt redefiniți la nivelul claselor derivate. Astfel, dacă constructorul clasei de bază are patru parametri pentru inițializare (coordonatele `x1`, `y1`, `x2`, `y2`), constructorul unui dreptunghi primeşte şi culoarea inițială *cul0*, iar cel pentru disc atât culoarea, cât şi stilul de haşură: `cul0`, `has0`.

```
program ObiecteGraficeDiverse;
uses Graph, Graph1, uMouse, Crt;
function Apartine(x,y,x1,y1,x2,y2: Integer): Boolean;
begin
    Apartine := (x1<=x) and (x<=x2) and (y1<=y) and (y<=y2)
end;
type TObiect = object
                 private
                   x1,y1,x2,y2: Integer;
                   procedure Display; virtual;
                   procedure Clear;
                 public
                   constructor Init(x10,y10,x20,y20: Integer);
                   procedure Move;
               end;
constructor TObiect.Init;
begin x1:=x10; y1:=y10; x2:=x20; y2:=y20; Display end;
procedure TObiect.Clear;
begin ClearView(x1,y1,x2,y2) end;
procedure TObiect.Display;
begin { va fi redefinita } end;
procedure TObiect.Move;
var b,px,py,x,y: Integer;
begin
```

```
  MouseData(b,px,py);
  if (b=1) and Apartine(px,py,x1,y1,x2,y2) then
     begin
       SetWriteMode(1); MouseHide;
       repeat
         MouseData(b,x,y);
         Rectangle(x1+x-px,y1+y-py,x2+x-px,y2+y-py);
         MouseShow; Delay(50); MouseHide;
         Rectangle(x1+x-px,y1+y-py,x2+x-px,y2+y-py);
       until b=0;
       SetWriteMode(0); Clear;
       x1:=x1+(x-px); y1:=y1+(y-py); x2:=x2+(x-px); y2:=y2+(y-py);
       Display; MouseShow
     end
end;
type TDrept = object(TObiect)
                private
                  cul: Byte;
                public
                  constructor Init(x10,y10,x20,y20,cul0: Integer);
                  procedure Display; virtual;
              end;
constructor TDrept.Init;
begin
  cul:=cul0;
  TObiect.Init(x10,y10,x20,y20)  {se apeleaza constructorul clasei de baza}
end;
procedure TDrept.Display;
begin
  SetColor(cul); Rectangle(x1,y1,x2,y2) { redefinirea metodei }
end;
type TDisc = object(TObiect)
               private cul,has: Byte;
               public
                  constructor Init(x10,y10,x20,y20,cul0,has0: Integer);
                  procedure Display; virtual;
            end;
constructor TDisc.Init;
begin
  cul:=cul0; has:=has0;
  TObiect.Init(x10,y10,x20,y20) {apel constructor clasa parinte}
end;
procedure TDisc.Display;
begin
  SetFillStyle(has,cul);
  FillEllipse((x1+x2) div 2,(y1+y2) div 2,(x2-x1) div 2, (y2-y1) div 2)
end;
const NrDreptunghiuri = 5; NrDiscuri = 3;
var Drp: array[1..NrDreptunghiuri] of TDrept;
    Dsc: array[1..NrDiscuri] of TDisc; i: Integer;
begin
  OpenGraph; SetBkColor(Blue); MouseInit; MouseShow;
  for i:=1 to NrDreptunghiuri do
      Drp[i].Init(10+50*i,100,50+50*i,100+10*i,8+i);
  for i:=1 to NrDiscuri do
      Dsc[i].Init(10+50*i,300,50+50*i,300+10*i,8+i,i);
  repeat
    for i:=1 to NrDreptunghiuri do Drp[i].Move;
    for i:=1 to NrDiscuri do Dsc[i].Move
  until KeyPressed; CloseGraph
end.
```

Observații

♦ Derivarea poate genera şi relații de moştenire mai "mari", în sensul că putem avea o clasă de obiecte `B` derivată dintr-o clasă părinte `A`, iar din `B` să se deriveze o clasă `C`. Atunci, `C` va moşteni şi de la `A`, iar `A` este un strămoş atât pentru `B`, cât şi, indirect, pentru obiectele de tip `C`.

♦ Dacă avem un obiect de tip `TObiect`, să zicem `var O: TObiect`, şi un obiect `var D: TDrept`, unde `TDrept` este derivat din `TObiect`, atunci putem realiza o aribuire de genul `O:=D`, nu şi invers: `D:=O`.

♦ Variabilele de tip obiect pot fi parametri pentru proceduri ca şi variabilele de orice tip. De asemenea, dacă un parametru formal într-o procedură este de un obiect strămoş, atunci procedurii i se poate trimite ca parametru efectiv un obiect descendent din acesta.

Rămâne la latitudinea cititorului ca să se folosească de toate avantajele pe care le oferă programarea orientată pe obiecte. Lucrările de la bibliografie notate cu semnul "%" pot fi consultate pentru o documentare mai bună în legătură cu programarea orientată pe obiecte şi posibilitățile oferite de această tehnică.

6. Un joc orientat-obiect. Utilizarea *Sound Blaster* -ului

Afişare rapidă

În paragraful acesta vom prezenta un simplu joc de îndemânare, realizat în modul text, deci folosind biblioteca `Crt`. Afişările vor fi făcute, însă, cu o procedură mai rapidă decât `Write`, cu procedura `PrintAt` pe care am inclus-o, alături de procedurile ce ascund, respectiv reafişează cursorul, într-o bibliotecă proprie `Crt1`, pe care o dăm ma jos.

Procedura `PrintAt` este mai rapidă decât `Write`, deoarece ea lucrează direct cu memoria video a calculatorului. Adresa de start a acesteia este `B800` pentru color, şi `B000`, pentru monocrom, în haxazecimal. Orice caracter se afişează într-o anumită linie şi coloană. Afişarea sa foloseşte doi octeți: primul reține codul `ASCII` al caracterului, iar cel de al doilea atributele sale grafice, adică: culoarea de fond, culoarea de scriere, intensitatea. Aceste informații sunt prelucrate rapid de procedura `PrintAt`, care va afişa un şir de caractere `S`, începând cu poziția `(x,y)`.

În rest, biblioteca `Crt1` conține procedurile de ascundere/reafişare a cursorului, `HideCursor`, respectiv `ShowCursor`.

```
unit Crt1;
interface
    procedure HideCursor;
    procedure ShowCursor;
    { PrintAt afiseaza rapid, direct in memoria video, un sir S,
      incepind cu coloana x si cu lina y,
      pe fondul ("background") b, de culoarea ("foreground") f }
    procedure PrintAt(x, y: Byte; S: String; f, b: Byte);
implementation
  uses Crt;
  procedure HideCursor; assembler;
  asm MOV AX,$0100; MOV CX,$2607; INT $10 end;
  procedure ShowCursor; assembler;
  asm MOV AX,$0100; MOV CX,$0506; INT $10 end;
  procedure PrintAt(x, y: Byte; S: String; f, b: Byte);
  type
   VideoLocation=record
```

```
                  { formatul unei locatii video pe doi bytes }
                  VideoData: Char; { caracterul afisat }
                  VideoAttribute: Byte;   { atributele (culorile) }
                end;
  var i: Byte; VideoSegment: Word; { locatia memoriei video }
      MonoSystem: Boolean;         { mono vs. color }
      VideoPtr: ^VideoLocation;    { pointer la locatii video }
  begin
  { Determina locatia de memorie unde va fi tiparit sirul, in
    concordanta cu tipul monitorului si cu locatia de pe ecran
    (coordonatele x si y). Apoi asociaza pointerul VideoPtr cu acea
    locatie: VidPtr este un pointer catre tipul VideoLocation.
    Insereaza un caracter si atributele sale, dupa care trece la
    noul caracter si la noua locatie video }
    MonoSystem := (LastMode in [0,2,7]);
    if MonoSystem then VideoSegment:=$B000 else VideoSegment:=$B800;
    VideoPtr := Ptr(VideoSegment,2*(80*(y-1)+(x-1)));
    for i := 1 to Length(S) do
        begin
          VideoPtr^.VideoAttribute := (b shl 4) + f;
          { high = fundal; low = culoarea de scris }
          VideoPtr^.VideoData := S[i];
          {pune caracterul la locatia respectiva}
          Inc(VideoPtr); { trece la urmatoarea locatie video }
        end
  end;
end.
```

Utilizarea *Sound Blaster*-ului

Ne dorim ca în timp ce vom juca jocul nostru, să avem puțină muzică. În acest sens, vom folosi un fişier text, '`MUSIC.TXT`', care va avea pe fiecare linie câte un număr, reprezentând numărul de ordine al unei note, dintr-o mulțime de note muzicale specificate în program.

Valoarea `255` corespunde unei pauze. Valorile din fişierul '`MUSIC.TXT`' sunt date la sfârşitul paragrafului.

Aceste valori vor fi folosite pentru a produce sunete, fie melodia ce constituie fondul muzical al jocului, fie diverse sunete de efect. În cazul cel mai general, sunetele vor fi produse cu ajutorul procedurii `Sound` din biblioteca `Crt`. Dacă însă dispunem de un calculator dotat cu o placă de sunet de tip `Sound Blaster` (pe 16 biți), atunci putem folosi cu succes canalele de *Sound Blaster* pentru a produce sunete. Biblioteca de mai jos cuprinde o serie de proceduri necesare trimiterii de sunete către *Sound Blaster*. Astfel, procedurile `SbOff` şi `SbReset` inițializează placa de sunet, iar `SbNote` trimite pe un anumit canal, o anumită notă (`On`), sau o opreşte (`Off`). Ea foloseşte procedura `SbOut` care trimite într-un registru de *Sound Blaster* o anumită dată. Procedurile se bazează pe `WriteAddr` şi `WriteData`, care sunt realizate în limbaj de asamblare. Dar aceasta contează, ca şi în cazul mouse-ului, mai puțin, mai important este cum să le utilizăm. În acest sens, programul - joc `Navele` este un exemplu.

```
unit uSBlast;
{ programarea SoundBlaster-ului }
interface
  procedure SbReset;
  procedure SbOff;
  procedure SbOut(r,d: Byte);
  procedure SbNote(channel: Byte;
note: Word; modd: Boolean);
```

```
  const On = True; Off = False;
implementation
  const MaxReg = $F5;
  procedure WriteAddr(b: Byte);
assembler;
  asm
    mov  al,b; mov  dx,388h;
    out  dx,al; mov  cx,6;
```

```
 @wait:
    in   al,dx; loop @wait
  end;
  procedure WriteData(b: Byte);
assembler;
  asm
    mov  al,b; mov  dx,389h;
    out  dx,al; mov  cx,35h;
    dec  dx;
 @wait:
    in   al,dx; loop @wait
  end;
  procedure SbReset;
  var i : Byte;
  begin
   for i := 1 to MaxReg do
       begin
         WriteAddr(i);
         WriteData(0)
       end
  end;
  procedure SbOff;
  begin
    WriteAddr($b0); WriteData($11)
  end;
  { r=registru, d=data }
  procedure SbOut(r,d: Byte);
  begin
    WriteAddr(r); WriteData(d)
  end;
  procedure SbNote(channel: Byte;
      note: Word; modd: Boolean);
 begin
   SbOut($a0+channel,Lo(note));
   SbOut($b0+channel,
         ($20*Byte(modd)) or $10
         or Hi(note))
 end;

begin
     SbReset
end.
```

Atenţie, suntem atacaţi !

Următorul program reprezintă un simplu joc de animaţie şi îndemânare, orientat-obiect, pe care vă propunem să-l "descifraţi" singur.

Mai multe nave "extraterestre", de aceeaşi formă, dar de culori diferite, se mişcă aleator în spaţiu, iar noi trebuie să le distrugem. În acest sens dispunem de o navă proprie, care se poate deplasa pe o linie de jos a ecranului, în stânga sau în dreapta, folosind săgeţile de cursor. Putem trage în duşmani cu tasta de spaţiu. Dacă am tras un glonte la un moment dat şi apăsăm din nou pe trăgaci, atunci glontele vechi dispare şi apare altul nou. Dacă întâlneşte nave inamice, acesta le distruge. Timpul nostru de acţiune este limitat. Dacă nu acţionăm cum trebuie, pierdem jocul, căci extratereştrii aterizează. Singura consolare va fi atunci doar muzica ce se aude în difuzorul intern sau în cele ale *Sound Blaster*-ului. Aşadar, succes !

```
program Navele;
{ joc orientat obiect cu melodie la SoundBlaster ! }
uses Crt, Crt1, uSBlast;
const
  kbUp=#72; kbDown=#80; kbLeft=#75; kbRight=#77;
  kbSpace=#32; kbAltS=#31; kbEsc=#27; kbF1=#59; Pauza1=2; Pauza2=2;
  Nota: array[0..47] of Integer =
  ({DO}131,139,{RE}147,155,{MI}164,{FA}175,185,{SOL}196,208,{LA}220,233,
  {SI}247,{DO}262,278,{RE}294,312,{MI}330,{FA}349,
  370,{SOL}392,416,{LA}440,467,{SI}494,{DO}523,555,{RE}587,623,{MI}659,
  {FA}698,741,{SOL}784,832,{LA}880,934,{SI}988,{DO}1047,1111,{RE}1175,
  1247,{MI}1319,{FA}1397,1482,{SOL}1568,1664,{LA}1760,1868,{SI}1976);
type Str3 = String[3];
     TNava = object
               x,y,dx,dy,cul: Byte; model: Str3;
               procedure Init(x0,y0: Byte; model0: Str3);
               procedure Display;
               procedure Clear;
               procedure Move;
             end;
var Nava: array[1..20] of TNava; NrNave: Integer;
    fisier: Text; frecv,frecv1: Byte;
    Melo: array[1..300] of Byte; contor,lung_melo: Integer;
```

```
type TOm = object
             c:integer;
             procedure Init(c0:integer);
             procedure Move;
           end;
var Om:TOm;
type TGlont = object
                l,c:integer;
                procedure Init(l0,c0:integer);
                procedure Move;
                procedure Clear;
              end;
var Glont:TGlont; SB, ExistaGlont, Sonor, Escape:boolean;
    i, Timp:Integer; sir: String;
procedure Beeps;
var i: Integer;
begin
  for i:=2 to 10 do
      begin
        if SB then SbNote(i,200+50*i,On) else Sound(200+50*i);
        Delay(5)
      end;
  if SB then SbReset else NoSound
end;
procedure TNava.Init;
begin
  x:=x0; y:=y0; model:=model0;
  dx:=Random(3)-1; dy:=Random(3)-1; cul:=9+Random(7)
end;
procedure TNava.Display;
begin PrintAt(x-1,y,model,cul,Black) end;
procedure TNava.Clear;
begin PrintAt(x-1,y,'   ',Black,Black) end; { 3 spatii }
procedure TNava.Move;
var i: Integer;
begin
  Clear; Inc(x,dx); Inc(y,dy);
  if x<2 then x:=2; if x>78 then x:=78;
  if y<2 then y:=2; if y>20 then y:=20;
  Display; Delay(Pauza2); dx:=Random(3)-1; dy:=Random(3)-1
end;
procedure TOm.Init(c0:integer);
begin
  c:=c0; PrintAt(c,21,'=³=',Yellow,Black)
end;
procedure TOm.Move;
var comanda:char;
begin
  if KeyPressed then begin
     if SB then SbNote(0,500,On) else Sound(200);
     Delay(1);
     if SB then SbNote(0,0,Off) else NoSound;
     Dec(Timp); comanda:=ReadKey;
     if comanda=kbEsc then Escape:=True
     else begin
             if comanda=kbSpace then
               if not ExistaGlont then begin
                   Glont.Init(20,c+1); ExistaGlont:=True end
               else begin Glont.Clear; Glont.Init(20,c+1) end;
             if comanda=#0 then begin comanda:=ReadKey;
                case comanda of
                     kbAltS: Sonor:=not Sonor;
```

```
                    kbLeft: begin PrintAt(c,21,'   ',Black,Black);
                            Dec(c); if c<1 then c:=1;
                            PrintAt(c,21,'=³=',Yellow, Black);
                            end;
                    kbRight: begin PrintAt(c,21,'   ',Black,Black);
                             Inc(c); if c>78 then c:=78;
                             PrintAt(c,21,'=³=',Yellow,Black);
                             end
               end {case}
            end {if 2}
         end {else}
      end {if 1}
end;
procedure TGlont.Init(l0,c0:integer);
begin c:=c0; l:=l0; PrintAt(c,l,#24,LightRed,Black) end;
procedure TGlont.Move;
var i:integer;
begin
      if ExistaGlont then begin
         PrintAt(c,l,' ',Black,Black); Dec(l);
         if l=0 then
            ExistaGlont:=False
         else begin
              PrintAt(c,l,#24,LightRed,Black);
              for i:=1 to NrNave do
                  if (l=Nava[i].y) and (abs(c-Nava[i].x)<=1) then
                     begin
                     Beeps; PrintAt(Nava[i].x-1,Nava[i].y,'   ',Black,Black);
                     Nava[i].y:=Nava[NrNave].y; Nava[i].x:=Nava[NrNave].x;
                     Dec(NrNave)
                     end
              end;
      end
end;
procedure TGlont.Clear;
begin PrintAt(c,l,' ',Black,Black) end;
procedure Cinta;
begin
      if contor=lung_melo+1 then
         contor:=1;
      frecv:=Melo[contor];
      if frecv<>255 then
         begin
           if SB then SbNote(1,10+nota[frecv],On)
           else Sound(nota[frecv]);
           Delay(40-NrNave);
           frecv1:=frecv;
         end
      else
          begin
            if SB then SbNote(1,10+nota[frecv1],Off)
            else NoSound;
            Delay(80-NrNave)
          end;
      Inc(contor)
end;
procedure CitesteMelodia(nume_fis: String);
begin
      contor:=0;
      Assign(fisier,nume_fis); Reset(fisier);
      while not eof(fisier) do
            begin
```

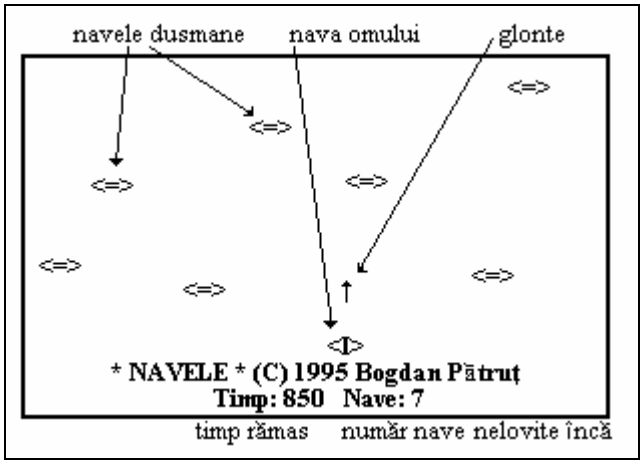
```
            Inc(contor);
            Read(fisier,Melo[contor])
          end;
     SbReset;
     lung_melo:=contor;
     contor:=1
end;

begin
     { mărirea vitezei de recepţionare a tastei apăsate }
     port[$60]:=$F3;delay(200);port[$60]:=0;
     TextColor(LightBlue); TextBackGround(0); ClrScr;
     WriteLn('Va prezentam jocul NAVE ! ');
     WriteLn; WriteLn('Sunet la SoundBlaster sau la Difuzor [s/d] ?');
     SB := ReadKey in ['s','S']; ClrScr; HideCursor; NrNave:=10;
     for i:=1 to NrNave do Nava[i].Init(Random(70)+5,Random(10)+5,'<=>');
     Om.Init(15); ExistaGlont:=False; Timp:=2000; Escape:=False;
     PrintAt(20,23,'* NAVE * (P), (C) 1995 Bogdan P†trut', LightCyan,Blue);
     CitesteMelodia('MUSIC.TXT');
     repeat
        Cinta; Dec(Timp); Str(Timp,sir);
        PrintAt(25,24,'Timp: '+sir+'    ',White,Black);
        Str(NrNave,sir); PrintAT(40,24,'Nave: '+sir+'    ',White,Black);
        Delay(Pauza1);
        Om.Move;
        for i:=1 to NrNave do
            begin
              Nava[i].Move; Om.Move
            end;
        if ExistaGlont then Glont.Move;
      Until (NrNave=0) or (Timp=0) or Escape;
      Close(fisier); SbReset;
      for i:=1 to NrNave do Nava[i].Clear;
      if Escape then PrintAt(25,10,'Joc terminat prin abandon ...',
                             Yellow,Blue)
      else
        if Timp=0 then
           PrintAt(20,10,'Timp epuizat. Invadatorii au invins !',Yellow,Blue)
             else
           PrintAt(22,10,'Tu ai cŒstigat . Felicit†ri ! ',Red,White);
      Delay(100); TextColor(7); TextBackground(0); ClrScr; ShowCursor
end.
```

Fişierul `MUSIC.TXT` conţine, câte una pe linie, următoarele valori întregi. Prin urmare, dumneavoastră veţi edita un fişier text `MUSIC.TXT`, în care după fiecare valoare scrisă, veţi apăsa tasta `Enter`. Firește, puteţi converti fişierul într-un fişier de tip `File of Byte`, pentru a face economie. Melodia vă este cunoscută dintr-un desen animat cu *Chilly Willy*.

24,24,23,23,24,24,0,0,12,12,0,0,12,12,0,0,19,19,17,17,16,16,19,19,24,24,0,0,24,24,28,28,26,26,25,25,26,26,2,2,
14,14,2,2,14,14,2,2,26,26,24,24,23,23,24,24,21,21,23,23,19,19,0,0,21,21,23,23,24,24,23,23,21,21,19,19,21,21,
19,19,17,17,16,16,17,17,16,16,14,14,12,12,11,11,255,255,255,255,255,255,7,7,9,9,11,11,12,12,14,14,16,16,17,
17,14,14,16,16,0,0,12,12,0,0,12,12,0,0,0,0,0,0,24,24,19,19,16,16,19,19,24,24,19,19,16,16,19,19,21,21,0,0,17,17,
0,0,17,17,4,4,2,2,0,0,26,26,21,21,17,17,21,21,26,26,21,21,17,17,21,21,23,23,0,0,19,19,0,0,19,19,0,0,21,21,23,
23,24,24,23,23,21,21,19,19,21,21,19,19,17,17,16,16,17,17,16,16,14,14,12,12,11,11,255,255,255,255,255,255,
7,7,12,12,11,11,14,14,12,12,16,16,14,14,17,17,16,16,0,0,12,12,0,0,12,12,0,0.

În figura următoare este dat un aspect din timpul rulării jocului `Navele`.



7. Exerciţii recapitulative

1. Problema bilelor. În cuprinsul primelor lecţii, am considerat următoarea problemă: pe o masă de biliard (ecranul cu `80×25` poziţii al calculatorului) se află mai multe bile, ce au culori diferite şi se mişcă pe diagonală, până se lovesc de una din margini. Atunci se reflectă sub un unghi egal cu cel incident şi îşi continuă deplasarea, tot aşa, până se apasă o tastă. Problema a fost rezolvată mai întâi folosind doi vectori care păstrau coordonatele bilelor, apoi un vector de înregistrări. Se cere să se simuleze mişcarea bilelor, folosind drept structură de date de reprezentare a lor un vector de obiecte.

2. Ferestre. Să se definească un tip obiect `TFereastra` care să permită administrarea unei ferestre grafice, conţinând o informaţie grafică de orice gen, un antet de unde poate fi mutată etc.. Se vor scrie metode pentru mutare, redimensionare, minimizare, maximizare, activare etc..

3. Liste - obiect. Să se realizeze o implementare obiectuală a unor liste dublu înlănţuite de şiruri de caractere, care să permită următoarele operaţii:

- iniţializarea listei, testarea dacă este vidă sau nu;
- adăugarea unui element la sfârşitul listei;
- inserarea unui element după cel curent;
- ştergerea elementului curent din listă;
- afişarea lungimii listei, a informaţiei elementului curent;
- poziţionarea la elementul următor, precedent;
- poziţionare pe primul şi pe ultimul element din listă, testarea acestor poziţii;
- ordonarea crescătoare şi afişarea listei.

BIBLIOGRAFIE

Notă: lucrările notate cu "%" sunt indicate cititorilor care vor să aprofundeze limbajul Turbo-Pascal, lucrările notate cu "*" sunt indicate celor care vor să aprofundeze tehnicile de programare şi algoritmica, în general, iar lucrările notate cu "#" se adresează cititorilor pasionați de grafica şi sunet pe calculator.

1(%). Valentin Cristea, Irina Athanasiu, Eugenia Kalisz, Lorina Negreanu - **Borland Pascal 7.0 pentru Windows**, Ed. Teora, Bucureşti, 1994.
2(%*). Doina Rancea - **Limbajul Turbo Pascal**, Ed. Libris, Cluj-Napoca, 1995.
3(#%). Sorin Tudor - **Turbo Pascal. Algoritmi şi limbaje de programare**, Ed. Teora, Bucureşti, 1995.
4(%). Ovidiu Pavel, Maria Bouruc - **Mediul de programare Pascal. Manualul limbajului**, Bucureşti, 1986.
5(%). Fl. Munteanu, T. Ionescu, Gh. Muscă, Daniela Tătaru, S. M. Dascălu - **Programarea calculatoarelor. Manual pentru licee de informatică, cl. X-XII**, Ed. Didactică şi Pedagogică, Bucureşti, 1994.
6(#%). Kovacs Sandor, **Turbo Pascal 6.0 - Ghid de utilizare**, Ed. MicroInformatica, Cluj-Napoca, 1991.
7(#%). Makkai Andras, Cecilia Crişan, Kovacs Sandor - **Ghid de utilizare Turbo Pascal 5.0-5.5**, Ed. Romanian Software Comp. (MicroInformatica), Cluj-Napoca, 1991.
8(#%). Winfried Kassera, Volker Kassera - **Programarea în Turbo Pascal 6.0**, Ed. Tipomur, Târgu Mureş, 1992.
9(#). Lucian Vasiu, Radu Grama, Alexandra Aldica - **Programare avansată în Turbo Pascal 6.0**, Ed. MicroInformatica, Cluj-Napoca, 1993.
10(*%). Grigore Albeanu - **Programarea în Pascal, Turbo Pascal. Culegere de probleme**, Ed. Tehnică, Bucureşti, 1994.
11(*#). Dorel Lucanu - **Proiectarea algoritmilor. Tehnici elementare**, Ed. Universității "Al. I. Cuza", Iaşi, 1993.
12(*). Adrian Atanasiu - **Concursuri de informatică. Probleme propuse**, Ed. Petrion, Bucureşti, 1995.
13(*). Victor Mitrana - **Provocarea algoritmilor. Probleme pentru concursurile de informatică**, Ed. Agni, Bucureşti, 1994.
14(*). Leon Livovschi, Horia Georgescu - **Sinteza şi analiza algoritmilor**, Ed. Ştiințifică şi Enciclopedică, Bucureşti, 1986.
15(*). Valentin Cristea, Irina Atanasiu. Eugenia Kalisz, Valeriu Iorga - **Tehnici de programare**, Ed. Teora, Bucureşti, 1993.
16(*). Sorin Tudor - **Tehnici de programare**, Ed. Teora, Bucureşti, 1994.
17(*). Cornelius Croitoru - **Tehnici de bază în optimizarea combinatorie**, Ed. Universității "Al. I. Cuza", Iaşi, 1992.
18(*). Ioan Tomescu - **Bazele informaticii. Manual pentru clasa a X-a**, Ed. Didactică şi Pedagogică, Bucureşti, 1994.
19(*). Clara Ionescu, Ioan Zsako - **Structuri arborescente cu aplicațiile lor**, Ed. Tehnică, Bucureşti, 1990.
20(#). Bogdan Pătruț - **Grafică în OOP ...şi nu numai...**, Ed. Adias, Râmnicu Vâlcea, 1995.
21(#). * * * - **Fido Pascal Conference**, BBS, 1994.
22(#). * * * - **FE - A stroke font editor**, Borland International, 1994.
23(#). * * * - **Autodesk FLI file format**, Autodesk, 1995.
24(*). * * * - **Info-Star** 1/1995, Ed. Libris, Cluj-Napoca, 1995.

INDEX

În continuare sunt dați cei mai utilizați termeni în lucrare, precum şi o serie de proceduri, programe etc. pe care le-am considerat mai importante.
Fiecare termen este dat împreună cu lecția şi paragraful în care apare.

Notații: *alg.* = algoritm, metodă de programare; *proc.* procedură, *func.* = funcție, *prog.* = program, *tip* = tip de date, *unit* = unit, bibliotecă.

A

admitere in liceu (prog.) 4.3, 5.2, 8.3
afisare in mod grafic 11.1, 11.3
afisare rapida 12.6
alg. clasic al lui Euclid 5.1
algoritm 1.1
animatie 11.6
antet functie 5.1
Apartine (func.) 11.4, 12.2
apel prin valoare/referinta 5.1
Append (proc.) 9.3
arbore 7.4, 8.4, 8.5, 8.6, 9.1
arbore binar 8.4, 8.5, 8.6
arbore de cautare-sortare 8.4
arbore partial de cost minim 7.4
arborescenta 7.4, 8.4
articol 1.5, 4.7
asm...end 5.1
assembler 5.2
Assign (proc.) 9.2
AssignCrt 9.4
atribut 12.1

B

backtracking (alg.) 6.4
bar (proc.) 11.1
bgi 11.1
biblioteca 10.1, 10.2, 11.5, 11.6, 12.6, 2.11, 9.1,
bloc 5.1
BlockRead (proc.) 9.5
BlockWrite (proc.) 9.5
boolean (tip) 2.6
bubble-sort (alg.) 4.3, 6.2, 9.3
Byte (tip) 4.1

C

calcul radical (alg.) 3.2
calculator electronic 1.4
camp 12.1
cautare binara 1.1
cautare secventiala 1.1
Char (tip) 2.5
Chr (func.) 4.5
CHR 11.1, 11.5
ciclu (elementar) 7.1
ciclu hamiltonian 7.1, 7.5
Circle (proc.) 11.1
citire in mod grafic 11.1, 11.3
clasa 12.1, 12.2
ClearDevice (proc.) 11.1
ClearWiew (proc.) 11.1, 12.2
ClearViewport (proc.) 11.1
cmmdc (alg.) 3.2, 3.4, 5.1
cmmmc (alg.) 3.2
coada 8.2
colorarea varfurilor unui graf 7.5
combinare 6.1
componente conexe 7.1
conditie de consistenta 5.4
const 2.1, 5.3
constanta 2.1
constructor 12.1, 12.5
conventie folosita in descrierea procedurilor 1.4
corp functie 5.1
CR, LF 9.3
CRT (unit) 2.11, 3.2, 10.1
CRT1 (unit) 12.6
culori 2.11, 11.1
cursor 10.3

D

Dec (proc.) 4.2
derivare 12.1, 12.3, 12.5
derivare formala (prog.) 8.5
derivare multipla 12.1
Desenator (prog.) 11.4
destructor 12.1
diamant 11.2, 11.4
digraf (graf orientat) 7.1
digraf conex 7.1
digraf tare conex 7.1
digraf unilateral conex 7.1
Dijkstra (alg.) 7.3
Dispose (proc.) 8.1
divide et impera (alg.) 6.6
DOS (unit) 9.1, 10.1
driver grafic 11.1
drum 7.1, 7.3

E

ecuatie de gradul 1 (prog.) 2.11
ecuatie de gradul 2 (prog.) 2.11
editor de animatie (prog.) 11.6
efect de senila 11.1
elementele programarii structurate 1.2
Ellipse (proc.) 11.1
EoF (func.) 9.3
EoLn (func.) 9.3
Erase (proc.) 9.3
evaluare functie intr-un punct 5.5
explorare in largime/adancime 7.1
expresie constanta numerica 1.3

F

fereastra 12.7
File 9.4, 9.5
FileSize (func.) 9.5
FillEllipse (proc.) 11.1
fisier 9.1, 9.2, 9.3, 9.4, 9.5, 10.3, 11.5
fisier cu tip 9.4
fisier fara tip 11.5, 11.6
fisier fara tip 9.5
fisier text 9.3
FLI 11.6
flic 11.6
font 11.5
forma poloneza inversa 8.5
FreeMem (proc.) 11.1, 11.6
frunza 8.4
functie 5.5, 11.3, 12.1
functionalitate 12.1

G

gandire algoritmica 1.1
GetImage (proc.) 11.1, 11.6
GetMaxColor (func.) 11.1
GetMaxX (func.) 11.1
GetMaxY (func.) 11.1
GetMem (proc.) 11.1, 11.6
gradul unui nod 7.1
graf 7.1
graf bipartit 7.1
graf complet 7.1
graf conex 7.1
graf conex 8.4
graf eulerian 7.1
graf hamiltonian 7.1
graf partial 7.1
graf suport 8.4
grafic functie 11.3
GRAPH (unit) 10.1, 11.1
GRAPH1 (unit) 11.1
greedy (alg.) 4.2, 6.3

H

Halt (proc.) 5.2
histograma 4.4

I,Î

identificator 2.1
ImageSize (func.) 11.1, 11.6
imprimanta 10.1, 10.3
Inc (proc.) 4.2
inchiderea tranzitiva a unei relatii binare 7.5
informatica 1.4
inordine (alg.) 8.4
inregistrare 1.5, 4.7
instructiune 1.1

instructiune compusa 1.2, 3.1
instructiune de atribuire (asignare) 1.3
instructiunea de salt neconditionat 1.2
Integer (tip) 2.3
interclasare vector (alg.) 4.3, 6.2, 6.6
interschimbarea a doua variabile 3.2

J

joc (oop) 12.6

K

Kruskal (alg.) 7.4

L

lant 7.1
lant elementar 7.1
lant eulerian 7.1
limbaj de programare 1.4
Line (proc.) 11.1
lista 7.1, 8.1, 8.3, 9.4, 12.7
lista de adiacenta 7.1
lista dublu inlantuita 8.3

M

matrice 4.2, 4.6, 7.1
matrice de adiacenta 7.1
meniu 11.4
metoda 12.1, 12.2, 12.4, 12.5
metoda virtuala (dinamica) 12.5
mod grafic 11.1, 11.4
mod text 6.5
Morphing (prog.) 11.6
morphing 11.6
mostenire 12.1
mouse 6.5, 10.3, 11.4, 12.2
MouseData 11.4
MouseHide 6.5, 11.4
MouseInit 6.5, 11.4
MouseMove 11.4
MouseShow 6.5, 11.4
MouseTData 6.5
muchie 7.1
multigraf 7.1
multime (tip) 4.8

N

New (proc.) 8.1
Nil 8.1
nod 7.1
numar prim (alg.) 3.2, 3.6
nume de variabila (indexata) 1.3
numere la intamplare (aleatoare) 3.2

O

obiect 12.1, 12.2, 12.7
object 12.1
OpenGraph (proc.) 11.1, 11.6
Ord (func.) 4.5
OutTextXY (proc.) 11.1

P

padure 7.4
ParamCount (func.) 9.3
parametri 5.1, 9.3
parametri formali si efectivi 5.1
ParamStr (func.) 9.3
partea declaratiilor locale 5.1
permutare 6.1
pixel 11.1
placa grafica 11.1
Pointer (tip) 11.1, 11.5
pointer 8.1
polimorfism 12.4
polinom 4.4
postordine (alg.) 8.4
preordine (alg.) 8.4, 9.1
prim (alg.) 7.4
primalitate numar (alg.) 3.2, 3.6
principiuul optimalitatii 6.7
PrintAt (proc.) 12.6
PRINTER (unit) 10.1
private 12.2
problema "Attila si regele" 6.8
problema damelor 6.4
problema labirintului 6.4, 6.5
problema numerelor pitagoreice 6.8
problema platourilor 6.7
problema raftului 6.7

problema rucsacului 6.3, 6.4
problema steagului national olandez 6.8
problema subsirului maximal 6.7
procedura 1.1, 5.1, 12.1
program 1.1
programare 1.2
programare dinamica (alg.) 6.7, 7.3
programare orientata pe obiecte (OOP, POO) 12.1
programare structurata 1.2
public 12.2
PutImage (proc.) 11.1, 11.6
PutPixel (proc.) 11.1

Q

quick-sort (alg.) 6.6, 9.4, 10.4

R

radacina 8.4
Read (proc.) 9.3
Real (tip) 2.4
Rectangle (proc.) 11.1
recursivitate 1.1, 5.1, 5.4, 11.2
recursivitate directa 5.4
recursivitate indirecta (incrucisata) 5.4
Rename (proc.) 9.3
reprezentare grafuri 7.1
Reset (proc.) 9.3
Rewrite (proc.) 9.3

S

schema logica 1.2
selectie directa (alg.) 4.3, 6.2
set (tip) 4.8
SetActivePage (proc.) 11.6
SetBkColor (proc.) 11.1
SetColor (proc.) 11.1
SetFillstyle (proc.) 11.1
SetViewport (proc.) 11.1
SetVisualPage (proc.) 11.6
SetWriteMode 11.1, 11.6
shell (alg.) 6.2
ShortInt (tip) 4.1
sir 1.5
sir de caractere 4.5
sortare 6.2, 6.6, 8.4, 9.3, 9.4
sortare prin interclasare (alg.) 6.2, 6.6
Sound Blaster 12.6
stiva 5.4, 5.5, 8.2, 8.5
String (tip) 2.7, 4.5
structura 1.5, 12.1
subgraf 7.1
subprogram 5.1
SYSTEM (unit) 10.1

T

tablou bidimensional 4.2
tablou unidimensional 4.2
tip 1.5, 2.2, 8.1, 9.1, 12.1, 12.2
tip ordinal (scalar) 2.3
tip pointer 8.1
tip referinta 8.1
tip rezultat functie 5.1
TPC 11.6
TransfSurface (proc.) 11.6
TURBO.TPL 10.1
turnurile din Hanoi 5.2

U

UFONT (unit) 11.5
UMOUSE (unit) 11.4
unit 2.11, 9.1, 10.1, 10.2, 12.6, 12.7
unitate de program 10.1
USBLAST (unit) 12.6
uses 10.1

V

var 7.1
variabila 1.3, 2.2, 5.3, 8.1, 9.1
variabila de tip subprogram 11.3
variabila dinamica 8.1
variabila statica 8.1
variabile de tip subprogram 5.3
variabile initializate 5.3
vecinatatea unui varf (nod) 7.1

vector 1.5, 4.2, 6.1, 10.2
vector caracteristic 6.1
virtual 12.5

W
Window (proc.) 4.7
Word (tip) 4.1

OPȚIUNI DE COMPILARE FOLOSITE

În programele din această carte s-au folosit următoarele opțiuni de compilare, în textele sursă:

{$X+} - Se precizează că funcțiile pot fi folosite şi ca proceduri, deci nu contează rezultatul întors de funcție, ci doar efectele ei laterale; în acest sens, un apel de genul ReadKey va aştepta acționarea unei taste, iar tasta apăsată nu contează.

{$I-}, {$I+} - După {$I-}, compilatorul nu va mai face verificarea reuşitei unei proceduri de intrare - ieşire, rămânând în sarcina programatorului acest lucru. În acest sens, se testează variabila de sistem IOResult, care, dacă nu este zero, indică prezența unei erori. Prin {$I+} se revine în modul obişnuit.

{$S-}, {$S+} - După {$S-}, compilatorul nu mai face verificări asupra depăşirii eventuale a stivei. Aceast\ opțiune se foloseşte în cazul programelor recursive, în care şirul de apeluri recursive este foarte mare, ceea ce umple stiva, însă suntem siguri că vor avea loc şi reîntoarcerile din aceste apeluri, deci golirea stivei. Prin {$S+} se revine la controlul normal al umplerii stivei.

{$F+} - Prin aceasta se precizează faptul că se poate defini un tip funcție sau procedură, şi variabile de acest tip. A se vedea paragraful 4 din lecția 5.